！！！！！ 面试题总结！！！！！

防抖节流

防抖

当点击搜索、添加之类的功能时，每点击一次都是发起请求，如果我们连续点击很多次，就会发起很多次请求，如果用户高频率的去触发一个事件，就会导致不良的后果出现不良的反应，性能差等问题，所以我们需要用到防抖，每次触发事件时设置一个延迟调用方法，并且取消之前的延时调用方法

防抖使当前函数在一定时间间隔里只执行一次，如果用户高频率的去触发，他会重新的去计算时间

优点：解决了数据实时变化实时请求导致的性能差流量浪费的问题
缺点：当用户在指定的时间间隔里一直操作，那么setTimeout里的函数永远不会执行

```
    <button id="btn">点我点我</button>

    let btn = document.querySelector('#btn');
    let timer = null;
    btn.addEventListener('click', function () {
        if (timer) {
            clearTimeout(timer); //timer延时器再次调用时触发 清空上一次的延时器
            timer = null
        }
        timer = setTimeout(() => {
            console.log('12345');
        }, 2000) //每隔两秒执行一次
    })
```

节流

高频时间触发，但是在n秒内只执行一次，所以节流会稀释函数的执行频率，就比如input输入框事件，只要输入就会触发，我们想让他隔一段时间触发就用到节流，当每次触发事件时，如果当前有等待执行的延时函数，则直接return

节流优化了函数触发频率过高导致的响应速度赶不上触发速度从而出现的卡顿、假死、延迟的现象

优点：避免项目中出现卡顿假死延迟等现象
缺点：如果用户一直操作，那么setTimeout里的函数会反复执行

```
    <input type="text" id="input">

    var input = document.querySelector('#input')

    input.addEventListener('input', throttle(handel, 2000)) //调用方法执行函数

    function throttle(fn, delay) { //封装方法
        let timer = null;
        let flag = true; //设置开关
```

```
        return function (...args) { //接受多个参数
            if (!flag) {  //如若发现了等待执行的延时函数
                return //直接return
            }
            flag = false; //开关为false
            timer = setTimeout(() => {
                fn.apply(this, args); //修改this指向  给到上面的函数
                flag = true //当开关为true时 两秒之后 开始执行
            }, delay)
        }
    }

    function handel(e) {
        console.log(e.target.value);
    }
```

相同点和不同点

相同点：都是用了setTimeout定时器在指定的时间间隔里执行函数，都是为了解决数据实时变化实时请求数据所导致的性能差问题
不同点：防抖是在同一个时间段里执行多次，就清除掉前面的事件执行最后一次，节流是在同一个时间段里如果没有超过指定时间间隔，就不去执行下一次

BFC

BFC是块级格式化上下文

解决的问题：margin重叠，两栏式布局，浮动元素的父级盒子高度塌陷

每个BFC区域都是独立隔绝的，互不影响，每个BFC区域只包括其子元素，不包括其子元素的子元素

触发条件：float除none以外的值，flex弹性布局，overflow除visible以外的值（hidden、auto、scroll）

发布订阅

发布订阅模式其实是对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于他的对象都将得到状态改变的通知
订阅者：把自己想订阅的事件注册到调度中心
发布者：发布事件到调度中心

优点：发布订阅模式将对象之间解耦，发布者不必关心消息传递给了谁，订阅者只需要负责对应消息的接收即可
缺点：创建订阅者本身需要消耗一定的时间和内存，多个发布者和订阅者嵌套在一起，难以跟踪和维护

实现思路：①定义一个变量存储这些事件 ②定义一个监听方法，用于把事件添加到事件变量中 ③定义一个删除方法，用于把是事件从变量中移除 ④定义一个触发方法，用于调用变量中的事件

```
    // 创建一个类
    class Observer{
        // 在类里添加一个构造函数
        constructor(){
            // 设置缓存列表（消息队列）
```

```
            this.message={
                // buy:[handlerA,handlerB]
            }
        }
        // 添加三个核心方法 $on(){} $off(){} $emit(){}
        /**
        * $on 向消息列表中添加内容
        * @param type 事件名（事件类型）
        * @param callback 回调函数
        */
        $on(type,callback){
            // 判断有没有这个事件类型
            if(!this.message[type]){
                // 如果没有这个事件类型，就初始化一个空的数组
                this.message[type] = []
            }
            // 如果有这个事件类型，就在后边push一个callback
            this.message[type].push(callback)
        }
        /**
        * $off 删除消息列表中的内容
        * @param type 事件名（事件类型）
        * @param callback 回调函数
        */
        $off(type,callback){
            // 判断消息队列中是否有这个事件类型，没有就直接return
            if(!this.message[type]) return
            // 判断是否有callback这个参数
            if(!callback){
                // 如果没有callback,就删掉整个事件
                this.message[type]=undefined
            }
            // 如果有callback,就仅仅删掉callback这个消息(过滤掉这个消息方法)
            this.message[type] = this.message[type].filter((item)=>item !==
callback)
        }
        /**
        * $emit 触发消息列表中的内容
        * @param type 事件名（事件类型）用来确定触发哪一个事件
        */
        $emit(type){
            // 判断是否有订阅
            if(!this.message[type]) return
            // 如果有订阅，就对这个type事件做一个轮询
            this.message[type].forEach(item=>{
                // 挨个执行每一个回调函数
                item()
            })
        }
    }
    // 创建一个实例
    const person1 = new Observer()

    // 向person1委托一些内容，调用person1的$on方法
    person1.$on('buy',handlerA)
    person1.$on('buy',handlerB)
    person1.$on('buy',handlerC)
```

```
    function handlerA(){
        console.log('handlerA');
    }
    function handlerB(){
        console.log('handlerB');
    }
    function handlerC(){
        console.log('handlerC');
    }

    // 测试删除一个消息
    // person1.$off('buy',handlerA)

    // 删除整个事件
    // person1.$off('buy')

    // 触发buy事件
    person1.$emit('buy')

    console.log('person1 >>',person1);
```

hooks

useState

useState是用来管理state的，他可以让函数组件具有维持状态的能力，即在函数组件的多次渲染之间，state是共享的
useState可以接收一个参数作为state的初始值，这个参数可以是任意数据类型，也可以是函数，返回一个数组，数组包括状态值和更新状态值的方法

useEffect

因为函数组件没有生命周期，useEffact帮助我们解决这个问题，useEffact有两个参数，可以通过传递不同的参数，来获得我们需要的类似于生命周期的作用
componentDidMount，componentDidUpdate 和 componentWillUnmount

当只有一个函数作为参数时，每次渲染完成都会触发
当有两个参数时，第一个参数是函数，第二个参数是空数组，只有页面第一次渲染成功时触发
当有两个参数时，第一个参数是函数，第二个参数是依赖项，只有依赖项改变了才能触发
当参数只有一个函数作为参数并且函数中返回了一个小函数时，卸载组件执行

useEffect和useLayoutEffect区别

useLayoutEffect是先知先觉，每次在dom实际更新以前被执行，里面可以再次setstate或者设置更新布局的监听回调

useEffect是后知后觉，在dom更新以后被执行。可以hold住大多数的场景

useCallback和useMemo

useCallBack与useMemo都会在第一次渲染时执行，之后会在其依赖的变量发生改变时再执行，都会返回缓存的值，useCallBack返回缓存的函数，useMemo返回缓存的变量

使用场景：比如说有一个父组件，其中包含子组件，子组件接收一个函数作为props；通常而言，如果父组件更新了，子组件也会执行更新；但是大多数场景下，更新是没有必要的，我们可以借助useCallback来返回函数，然后把这个函数作为props传递给子组件；这样，子组件就能避免不必要的更新。

useRef

useRef是一个方法，且useRef返回一个可变的ref对象
获取DOM节点，缓存值

自定义hook

1.自定义hook中可以调用其他hook
2.必须以use开头，就像组件必须以大写字母开头一样
3.自定义hook中管理state也是使用useState、useEffect，因为useState在调用的时候就是完全独立的
解决的问题：在编写组件的过程中又有一些逻辑是可以复用的，但是这些逻辑不需要在ui上展示或是有些组件不能在更细的拆分，可以将这些逻辑抽象成自定义hook

二叉搜索树

二叉搜索数又称二叉查找树，二叉排序树，二叉搜索树的查找效率非常的高，每个根节点最多只有两个叶子节点
特性:左子节点的值要比根节点小，右子节点的值要比根节点大

重绘重排

重绘

重绘就是布局计算完毕后，页面会重新绘制，这时浏览器会遍历渲染树，绘制每个节点，当元素外观变化但没有改变布局的时候，重新把元素绘制的过程。visibility、outline、背景色等属性的改变都能触发重绘

重排

重排就是浏览器在第一次渲染完页面布局以后，后续引起页面各节点位置重新计算或者重新布局的行为。因为元素的位置或者尺寸发生了变化，浏览器会重新计算渲染树，导致渲染树的一部分或者全部发生变化，需要重新绘制页面上影响的元素。改变页面尺寸，涉及元素尺寸或位置的操作都能触发重排

重绘就是改变某个节点的样式，重排就是增加或者删掉某些节点
重绘不一定触发重排，但重排一定会触发重绘

减少重绘重排

1.分离读写操作
```
var curLeft=div.offsetLeft; div.style.left=curLeft+1+'px';
```
2.集中改变样式 使用display:none ，不使用visibility，也不要改变它的 z-index
3.脱离文档流:浮动float,定位position:absolute,fixed

call,bind,apply

Symbol() 定义一个属性名，不会与方法里边的属性产生冲突

```
let obj = {
    name: '张三',
    fs: 'this is a JS'
}
let objs = {
    name: '李四'
}
function fun(a, b, c, d, e) {
    console.log(this);
    console.log(a, b, c, d, e);
}
```

call ()

call() 修改this指向，让函数立即执行，第一个参数就是this指向，第二个以及后面的参数就是函数的实参

```
Function.prototype.MyCall = function (thisArr,...arr){
    thisArr = (thisArr && thisArr instanceof Object) ? thisArr : window
    let fs = Symbol('fs')
    thisArr[fs] = this
    let res = thisArr[fs](...arr)
    delete thisArr[fs]
    return res
}
fun.MyCall(obj,111,222,333)
```

apply()

apply() 修改this指向，让函数立刻执行，第一个参数就是this指向，第二个以及后面的参数以数组的形式为实参

```
Function.prototype.MyCall = function (thisArr,arr){
    thisArr = (thisArr && thisArr instanceof Object) ? thisArr : window
    let fs = Symbol('fs')
    thisArr[fs] = this
    let res = thisArr[fs](...arr)
    delete thisArr[fs]
    return res
}
fun.MyCall(obj,[444,555,666])
```

bind()

bind() 修改this指向，不会让函数立刻执行，会返回新的函数，第一个参数就是this指向，第二个以及后面的参数就是函数的实参

```
    Function.prototype.MyBind = function (thisArr, ...arr) {
        thisArr = (thisArr && thisArr instanceof Object) ? thisArr : window
        let fs = Symbol('fs')
        thisArr[fs] = this
        return function (...arr1) {
            let arrAll = [...arr, ...arr1]
            let result = thisArr[fs](...arrAll)
            return result
        }
    }
    let fn = fun.MyBind(obj, 111, 222)
    fn(111, 222, 333)
```

js原生实现Promise方法

否非噢的

解决问题：回调地狱
状态：pending（正在进行中），fulfiled（已完成的），rejected（失败的）
方法：

- Promise.then
- Promise.catch（发生错误时触发的回调函数）
- Promise.finally（不管最后状态如何，都会执行的操作）
- Promise.all（全部成功结束，一起返回）
- Promise.allSetted（不管是成功状态还是失败状态，全部结束，一起返回）
- Promise.any（谁先成功，谁先返回 ）
- Promise.race（不管成功或者失败，谁先结束就返回谁）

实例方法 .then() .catch() .finally()
类方法api all allsetted race
promise三个状态: fulfilled 完成状态,rejected 错误状态,pending 等待状态

Promise 构造函数包含一个参数和一个带有 resolve（解析）和 reject（拒绝）两个参数的回调。在回调中执行一些操作（例如异步），如果一切都正常，则调用 resolve，否则调用 reject。

（1）Promise.prototype.then 方法返回的是一个新的 Promise 对象，因此可以采用链式写法。（第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。如果前一个回调函数返回的是Promise对象，这时后一个回调函数就会等待该Promise对象有了运行结果，才会进一步调用。）

（2）Promise.prototype.catch 方法是 Promise.prototype.then(null, rejection) 的别名，用于指定发生错误时的回调函数。（Promise 对象的错误具有"冒泡"性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个 catch 语句捕获。）

（3）Promise.all 方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。Promise.all 方法接受一个数组作为参数（例如：p1,p2,p3），只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

（4）Promise.race 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的Promise实例的返回值，就传递给p的返回值。

实现思路：1.构造函数 2.回调函数的参数 resolve reject 3.链式调用.then .catch

```
        function MyPromise(fn) {
            this.status = 'pending'
```

```
            this.value = undefined
            const _this = this
            function resolve(value){
                if(_this.status==='pending'){
                    _this.status = 'fulfilled'
                    _this.value=value
                }
            }
            function reject(value){
                if(_this.status==='pending'){
                    _this.status='rejected'
                    _this.value=value
                }
            }
            fn(resolve,reject)
        }

        MyPromise.prototype.then = function (onFulfilled, onRejected) {
            const _this = this
            return new MyPromise((resolve, reject) => {
                if (_this.status === 'fulfilled') {
                    onFulfilled(_this.value)
                }
                if (_this.status === 'rejected') {
                    onRejected(_this.value)
                }
                if (_this.status === 'pending') {
                    setTimeout(() => {
                        onFulfilled && onFulfilled.length !== 0 &&
onFulfilled(_this.value) && resolve(_this.value)
                        !onFulfilled && onRejected && onRejected.length !== 0 &&
onRejected(_this.value) && reject(_this.value)
                    }, 0)
                }
            })
        }

        MyPromise.prototype.catch = function (onFulfilled, onRejected) {
            return this.then(null, onRejected)
        }

        MyPromise.prototype.finally = function (onFinally) {
            return this.then((value) => {
                onFinally()
                return value
            }, (value) => {
                onFinally()
                return value
            })
        }

        const Ps = new MyPromise((resolve, reject) => {
            setTimeout(() => {
                resolve('成功') //fulfilled
                // reject('失败')
            }, 0)
        })
        const pp = Ps.then(null, (err) => { console.log(err); })
```

```
        pp.finally(function () {
            console.log('Finally');
        })
```

js原生实现 Promise.all,Promise.race,Promise.allSettled

Promise.all

```
Promise.all 提供了并行执行异步操作的能力，并且在所有异步操作执行完后才执行回调。可以将多个 Promise 实例，包装成一个新的 Promise 实例
Promise 接收一个对象作为参数，并且只返回一个Promise实例，返回结果是一个数组，数组包含了所有的回调结果。
```

```
    Promise.all = function(arr){
        let index=0; let result =[]
        return new Promise((resolve,reject)=>{
            arr.forEach((p,i)=>{
                Promise.resolve(p).then(val=>{
                    index++
                    result[i] = val
                    if(index === arr.length){
                    resolve(result)
                    },error=>{
                        reject(error)
                    }
                })
            })
        })
    }
```

Promise.allSettled

```
Promise.allSettled 可以获取数组中每个 promise 的结果，无论成功或失败
```

```
    Promise.allSettled=function(arr){
        let result = []
        return new Promise((resolve,reject)=>{
            arr.forEach((p,i)=>{
                Promise.resolve(p).then(val)=>{
                    result.push({
                        status:'fulfilled',
                        value:val
                    })
                    if(result.length === arr.length){
                        resolve(result)
                    }
                },err=>{
                    result.push({
                        status:'rejected'
                        reason:err
                    })
                    if(result.length===arr.length){
```

```
                        resolve(result)
                    }
                }
            })
        })
    }
```

Promise.race

```
Promise.race 返回最快的promise的结果
```

```
    Promise.race = function(arr){
        return new Promise((resolve,reject)=>{
            arr.forEach((p,v)=>{
                Promise.resolve(p).then(val){
                    resolve(val)
                },err=>{
                    reject(err)
                }
            })
        })
    }
```

js实现Curry函数

```
函数柯里化的意思是将多个参数的函数变成一个参数的函数

实现思路：通过函数的 length 属性获取函数的形参个数，形参的个数就是所需参数的个数

function beforeCurry(a,b,c,d,e){
    return a+b+c+d+e;
}

function curry(fn){
    // console.log(fn.length);
    if(fn.length<=1){
        return fn
    }
    const generator = (...args)=>{

        if(args.length === 2){
            // ... ...
        }

        if(fn.length === args.length){
            return fn(...args)
        }else{
            return (...args2)=>{
                return generator(...args,...args2)
            }
        }
    }
    return generator
}
```

```
const afterCurry = curry(beforeCurry)

// const result = afterCurry(1)(2)(3)(4)(5)
const breakFn = afterCurry(1)(2)

// 其他的事情

const result = breakFn(3)(4)(5)
console.log(result);
```

```
    function beforeCurry(a,b,c,d,e){
        return a+b+c+d+e
    }
    function curry(fn){
        function aaa(...args){
            if(fn.length === args.length){
                return fn(...args)
            }else{
                return (...args2)=>{
                    return aaa(...args,...args2)
                }
            }
        }
        return generator
    }
    const afterCurry = curry(beforeCurry)
    const result = afterCurry(1)(2)(3)(4)(5)
    console.log(result);
```

原型和原型链

所有构造函数都有一个原型：prototype

当访问一个对象的某个属性时，会先在这个对象本身属性上查找，如果没有找到，则会去它的__proto__隐式原型上查找，即它的构造函数的prototype，如果还没有找到就会再在构造函数的prototype的__proto__中查找，这样一层一层向上查找就会形成一个链式结构，我们称为原型链。

LRU

假如我们有一块内存，专门用来缓存我们最近访问的网页，访问一个新网页，我们就会往内存中添加一个网页地址，随着网页的不断增加，内存存满了，这个时候我们就需要考虑删除一些网页了。这个时候我们找到内存中最早访问的那个网页地址，然后把它删掉。这一整个过程就可以称之为 LRU 算法。

LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

```
// get和set用来获取和添加数据
// get获取数据后,该条数据更新到最前面
```

```
    class LRUCache{
        constructor(length){
            this.length=length //存储数据条数
            this.data=new Map() //仓库
```

```
        }
        set(key,value){
            const data = this.data
            if(data.has(key)){‘
                data.delete(key)
            }
            data.set(key,data)
            if(data.size>this.length){
                let del = data.keys(key).next().value
                data.delete(del)
            }
        }
        get(key){
            const data = this.data
            if(!data.has(key)){
                return null
            }
            const value = data.get(key)
            data.delete(key)
            data.set(key,value)
        }
    }
    let lrucache = new LRUCache(5)
    lrucache.set('name1',"张三1")
    lrucache.set('name2',"张三2")
    lrucache.set('name3',"张三3")
    lrucache.set('name4',"张三4")
    lrucache.set('name5',"张三5")
    lrucache.set('name6',"张三6")
    // lrucache.set('name2',"张三2")
    lrucache.get('name3')
    console.log(lrucache);
```

继承

组合寄生式继承：通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。
流程：使用寄生式继承来继承超类型的原型。将结果指定给子类型的原型。

原型链继承 （子类的原型等于父类的实例）借用构造函数继承 组合式继承 组合寄生式继承

原型链继承(子类无法向父类传参数，父类也没办法接到)

```
    function A(names) {
      this.names = names
    }
    function B() {

    }
    B.prototype = new A("11")
    let b = new B()
    console.log(b.names)
```

借用构造函数继承

```
function A(names) {
  this.names = names
}
function B(names) {
  A.call(this, names)
}
let res = new B("111")
console.log(res.names)
```

组合式继承

```
function A(names) {
  this.names = name
}
function B(names) {
  A.call(this, names)
}
B.prototype = new A()
let b = new B("1111")
console.log(b.names)
```

组合寄生式继承(通过构造函数来继承属性)

```
function A(names) {
  this.names = names
}
A.prototype = {
  sayName: function () {
    console.log(this.names)
  }
}
function B(names) {
  A.call(this, names)
}
B.prototype = Object.create(A.prototype)
let res = new B("11")
res.sayName()
```

es5继承及优缺点

1.构造函数继承

在子类构造函数中调用父类构造函数，可以理解为子类把父类的构造函数拷贝了一份到自己的作用域中，这个过程可以通过调用call或apply方法实现

优点：可以向父类构造函数传参数，并且每一个子类实例都有父类属性和方法的副本

缺点：正是因为这种方法调用父类，所以也只能继承父类的属性和方法，且继承的方法也无法复用。并且不能继承父类原型的属性和方法。（子类原型并未链接父类原型）

2.原型继承

借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型

优点：在没有必要兴师动众的创建构造函数，而只想让一个对象与另外一个对象保持类似的情况下，原型式继承是完全可以胜任的。

缺点：包含引用类型值的属性始终都会共享相同的值，并且这种方法有局限性。

3.寄生式继承(最稳定)

创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真的是他做的所有工作一样返回对象。

优点：在主要考虑对象，而不是自定义类型和构造函数的情况下，寄生式继承也是一种可选方案。

缺点：和原型继承类似有一定局限性。

4.组合式继承

他是原型链继承和构造函数继承组合，结合两种方式消除缺点发挥各自的长处，父类定义实例属性，并用构造函数继承让子类继承。父类原型定义可复用的方法或属性，并用原型链继承让子类继承。 这样既可以在原型上定义方法实现了函数复用，又能够保证每个实例都有他自己的属性。

优点：组合继承避免了原型链和借用构造函数的缺陷。融合了他们的优点成为JavaScript中最常用的继承模式。而且instanceof 和 isPrototypeOf() 也能够用于识别基于组合继承创建的对象。

缺点：组合继承调用了两次父类，因此子类实例和子类的原型上各存有一份父类实例的属性和方法。

js原生实现new方法

实现思路：创建一个空对象，获取构造函数，链接到原型，绑定this值，使用apply，将构造函数中的this指向新对象，最后返回新对象

```
    function create () {
        //创建一个空对象
        let obj = new Object();
        //获取构造函数  // let args = [].slice.call(arguments); let Fun = args.shift();
        let Constructor = [].shift.call(arguments);
        //链接到原型
        obj.__proto__ = Constructor.prototype;
        //绑定this值，使用apply，将构造函数中的this指向新对象，这样新对象就可以访问构造函数中的属性和方法
        let result = Constructor.apply(obj,arguments);
        //返回新对象
        return typeof result === "object" ? result : obj; //如果返回值是一个对象就返回该对象，否则返回构造函数的一个实例对象
    }
```

单例模式实现弹框

单例模式：创建型模式，提供了一种创建对象的最佳方式，他负责创建自己的对象，并确保只有单个对象被创建
要点：1.构造函数私有化(private) 2.静态的单实例
分为：1.懒汉模式（懒加载）需要的时候才去创建的，如果单例已经创建，再次调用获取接口将不会重新创建新的对象，而是直接返回之前创建的对象 2.饿汉模式（预加载）类加载的时候初始化

策略模式表单验证

策略模式：该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

跨域

1.jsonp：只能发起get请求，也只能解决get请求的跨域问题，会泄露用户信息,利用script标签的src属性来进行跨域
2.proxy：本质上利用了 http-proxy-middleware http 代理中间件，实现将请求转发给其他服务器。通过 proxy 实现代理请求后，会在浏览器与服务器之间添加一个代理服务器，本地发送请求时，中间代理服务器接收后转发给目标服务器，目标服务器返回数据，中间代理服务器将数据返回给浏览器。中间代理服务器与目标服务器之间不存在跨域资源问题。
3.cors：（后端）
4.nginx： 反向代理 利用nginx反向代理把跨域改为不跨域，支持各种请求方式

同源和跨域

同源：协议，域名，端口都相同
跨域：资源所在的服务器与资源内部发起请求的目标服务器不同源
解决跨域：1.jsonp 2.服务器代理 3.跨域跨域共享

原生实现jsonp

jsonp：用来解决跨域问题，

流程

流程：封装一个函数，参数为请求的地址，创建script标签并添加src属性，添加标签到页面上，节点一添加就会立马请求，回调函数，返回数据，最后页面加载好时方便调用

原理

jsonp原理：1.jsonp是用来解决跨域的一种方式 2.请求时不使用ajax，而是用script标签的src属性代替 3.只能发起get请求，也只能解决get请求的跨域问题 4.接收的数据格式比较单一

实现链表

定义

链表：数据的一种存储结构，一个链表包含若干个节点，每个节点至少包含一个数据域和指针域

分类

链表分为：单向链表和双向链表
    单向链表：每个节点中只包含一个数据域和指向下一个节点的指针域（next）
    双向链表：每个节点都有一个指向其前一个节点的指针域（prev）和其下一个节点的指针域（next）

实现流程

创建链表的流程思路：1.在创建链表时需要创建两个类：指针类和节点类 2.定义一个单向链表类 3.定义三个属性，一个记录单链表的长度或节点个数，一个记录链表的起始地址，一个记录当前节点 4.获取链表的长度 5.判断链表是否为空 6.遍历链表，不重复的话访问链表中的每个节点 7.如果当前节点不为空，则表明当前节点中存在数据，就让当前节点的指针指向下一节点 8.如果当前节点的下一个节点不为空，就拼接起来，看起来像一个链表 9.添加元素的话，先找到链表的最后一个节点，创建一个新节点，把新的节点放在链表里面去 10.删除节点的话，先遍历，找到需要删除节点的位置，找到该节点的上一个指针，找到后将该节点的上一个指针next改成指向下一个节点

排序

冒泡排序，快速排序，插入排序，选择排序，堆排序

冒泡排序

算法思想：判断两个相邻元素，大于则交换位置
算法步骤:从数组中第一个数开始，依次与下一个数比较并次交换比自己小的数，直到最后一个数。如果发生交换，则继续下面的步骤，如果未发生交换，则数组有序，排序结束，此时时间复杂度为O(n)；
每一轮”冒泡”结束后，最大的数将出现在乱序数列的最后一位。
算法平均复杂度：n(n^2)

```
    function sortArr(arr) {
        //外层循环，控制趟数，每一次找到一个最大值
        for (var i = 0; i < arr.length; i++) {
            // 内层循环,控制比较的次数，并且判断两个数的大小  -i 是因为每找到一个最大数 后面就不需要在比较了
            for (var j = 0; j < arr.length - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    var temp = arr[j]
                    arr[j] = arr[j + 1]
                    arr[j + 1] = temp
                }
            }
        }
        return arr
    }
```

快速排序

算法思想：取一个基准值，比基准值小的在左边，大的在右边；左右在继续这样的操作，最后合并。
算法步骤：从待排序的n个记录中任意选取一个记录（通常选取第一个记录）为分区标准；
把所有小于该排序列的记录移动到左边，把所有大于该排序码的记录移动到右边，中间放所选记录，称之为第一趟排序
然后对前后两个子序列分别重复上述过程，直到所有记录都排好序。
算法平均复杂度：n(n log n)

```
    function quick_sort(arr) {
        if (arr.length <= 1) {
            return arr
        }
        var quickIndex = Math.floor(arr.length / 2)
        var mid = arr.splice(quickIndex, 1)[0]
        var left = []
        var right = []
        for (var i = 0; i < arr.length; i++) {
```

```
            if (arr[i] > mid) {
                left.push(arr[i])
            } else {
                right.push(arr[i])
            }
        }
        return quick_sort(right).concat([mid], quick_sort(left))
    }
```

选择排序

算法思想：从所有记录中选出最小的一个数据元素与第一个位置的记录交换；然后在剩下的记录当中再找最小的与第二个位置的记录交换，循环到只剩下最后一个数据元素为止。
算法平均复杂度：n(n^2)

```
    function selection_sort(arr) {
        var minIndex
        var temp
        var len = arr.length
        for (var i = 0; i < len; i++) {
            minIndex = i
            for (var j = i + 1; j < len; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j
                }
            }
            temp = arr[i]
            arr[i] = arr[minIndex]
            arr[minIndex] = temp
        }
        return arr
    }
```

插入排序

算法思想：从待排序的n个记录中的第二个记录开始，依次与前面的记录比较并寻找插入的位置，每次外循环结束后，将当前的数插入到合适的位置。
算法平均复杂度：n(n^2)

```
    function insertion_sort(arr) {
        for (var i = 1; i < arr.length; i++) {
            var key = arr[i];
            var j = i - 1;
            while (arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = key;
        }
        return arr;
    }
```

栈堆

```
栈：先进后出，后进先出
堆：无序的键值对，存储方式跟顺序没有关系，不局限出入口，使用key取出value
队列：先进先出
```

！！！！！ vue/react！！！！！

生命周期

vue

```
    beforeCreate ：创建完成之前，此时不能访问data和methods，不能获取dom元素
    created：创建完成之后，此时可以访问data和methods，但是不能获取dom，一般在此时加载首屏数据，如果数据量庞大，会造成白屏
    beforeMount：挂载之前，此时不能获取dom元素
    mounted：挂载之后，dom已经完全渲染到页面上，可以获取dom元素
    beforeUpdate：更新之前，数据修改并且在视图上更新
    updated：更新之后
    beforeDestroy：组件销毁之前
    destroyed：组件销毁之后，此时不能获取dom节点
```

react

```
    初始化阶段（当组件实例被创建并插入 DOM 中时）：
        constructor：组件挂载之前，会调用它的构造函数
        getDerivedStateFromProps：在初始挂载及后续更新时都会被调用
        render:
        componentDidMount：组件挂载之后

    更新阶段（每当组件的 state 或 props 发生变化时）：
        getDerivedStateFromProps：在初始挂载及后续更新时都会被调用
        shouldComponentUpDate：当 props 或 state 发生变化时，
shouldComponentUpdate() 会在渲染执行之前被调用。
        render：
        getSnapshotBeforeUpdate： 在最近一次渲染输出之前调用，真实的dom构建完成，但还没渲染到页面中时
        componentDidUpdate：组件更新完调用

    销毁阶段（当组件从 DOM 中移除时）：
        componentWillUnmount：组件卸载及销毁之前调用
```

react被遗弃的生命周期

```
componentWillMount():dom渲染之前
componentWillUpdate():数据更新之前
componentWillReceiveProps():参数是最新的Props
```

组件通讯方式

vue

1.父传子：通过自定义属性，子组件通过props接收
2.子传父：通过自定义事件，子组件通过this.$emit(事件名，参数)接收
3.事件总线：$emit去监听，用$on去触发，注意需要$off来取消监听，否则可能会造成内存泄漏
4.$parent，$children:访问组件实例，进而获取或者改变父子组件的值
5.$ref:通过引用的方式获取子节点，常用于父组件中调用子组件的方法或者获取子组件的属性
6.vuex:集中式存储管理所有组件的状态
7.$attrs，$listener：适用于多级组件嵌套，但是不做中间处理的情况

react

1.父传子：通过自定义属性，子组件通过props接收
2.子传父：通过自定义事件，子组件通过this.$emit(事件名，参数)接收
3.跨组件（context上下文）：通过React.createContext生成上下文context，context中有两个组件分别是Provider和Consumer，Provider提供数据，Consumer接收数据
4.兄弟组件：在父组件定义好回调函数接收组件A传过来的值，组件A通过函数回调传值给父组件，父组件接收到组件A的值，再把接收的值传给组件B

类组件和函数组件的区别

1.类组件基于面向对象，他有继承，有状态属性和生命周期
2.函数组件基于FP，没有this指向，没有状态属性，没有生命周期
3.类组件优化：PureComponent，函数组件优化：React.memo

数据双向绑定

vue通过v-model实现数据双向绑定，v-on：input，v-bind：value

双向绑定原理：双向数据绑定是通过 数据劫持 结合 发布订阅模式的方式， 通过Object.defineProperty()来劫持各个属性的setter,getter,在数据变动时发布消息给订阅者，触发相应的监听回调。

Vue全家桶

vue-cli:脚手架,主要用来构建项目
vue-router:路由管理器，用于页面之间的跳转，路由守卫，路由懒加载等功能
vuex:状态管理模式,采用集中式存储管理应用的所有组件的状态,解决多组件数据通信

vuex

包含

（1）state（数据）：用来存放数据源，就是公共状态;
（2）getters（数据加工）：有的时候需要对数据源进行加工，返回需要的数据;
（3）actions（事件）：要执行的操作，可以进行异步事件
（4）mutations（执行）：操作结束之后，actions通过commit更新state数据源
（5）modules：使用单一状态树，致使应用的全部状态集中到一个很大的对象，所以把每个模块的局部状态分装使每一个模块拥有本身的 state、mutation、action、getters、甚至是嵌套子模块

流程

（1）通过dispatch去提交一个actions，
（2） actions接收到这个事件之后，在actions中可以执行一些异步|同步操作，根据不同的情况去分发给不同的mutations
（3）actions通过commit去触发mutations，
（4）mutations去更新state数据，state更新之后，就会通知vue进行渲染

vuex实现数据持久化

持久化原理：结合本地存储做到数据状态持久化，使页面刷新后数据不会初始化为默认状态。

将state中的数据，保存到本地存储中

```
npm i vuex-persistedstate -S

import createPersistedState from 'vuex-persistedstate'

const store = new Vuex.store({
    plugins:[createPersistedState()]
})
```

React全家桶

```
react:react核心
react-router:路由管理器,提供核心的路由组件与函数
    拆分成四个包:
        react-router
        react-router-native:提供了运行环境（浏览器与react-native）所需的特定组件
        react-router-dom:提供了运行环境（浏览器与react-native）所需的特定组件
        react-router-config:用来配置静态路由
redux:状态管理模式,相当于一个数据库
```

redux

组件

store：用来存储数据
reducer：真正的来管理数据
action：创建action，交由reducer处理
view：用来使用数据

流程

思到 瑞丢色 为噢

（1）store通过reducer创建了初始状态
（2）view通过store.getState()获取到了公共状态 艾珂神司
（3）view要改变状态，调用了actions 的方法，actions内部通过调用store.dispatch方法传递行为标识给reducer中,reducer接收到action并根据行为标识改变状态，返回新的状态
（4）store中的状态被reducer更改为新的状态，store.subscribe方法里的回调函数会执行，此时就可以通知view去重新获取state、

瑟不思哥ruai波

redux中间件

```
redux-logger:打印日志
redux-thunk:更好地管理异步编程处理（处理异步，多次派发dispatch）
redux-promise:允许action是一个promise
```

vue2.0和vue3.0的区别

区别

```
vue2.0中不管数据多大，都会在一开始就为其创建观察者；当数据很大时，这可能会在页面载入时造成明显的性能压力
vue3.0只会对"被用于渲染初始可见部分的数据"创建观察者，而且vue3.0的观察者更高效

vue -V 查看本地 vue 版本
1.下载安装 npm install -g vue@cli
2.创建项目文件：
    3.0：vue create project //项目的名称
    2.0：vue init webpack
3.启动项目
    3.0：npm run serve
    2.0：npm run dev
4.删除了vue list，移除了配置文件目录，config 和 build 文件夹
5.移除了 static 文件夹，新增 public 文件夹，并且 index.html 移动到 public 中
6.在 src 文件夹中新增了 views 文件夹，用于分类 视图组件 和 公共组件
```

vue3优势

```
更快、更小、更易维护、更易于原生、让开发者更轻松
1.更快
    放弃 Object.defineProperty ，使用更快的原生 Proxy
    基于 Proxy 观察者机制以满足全语言覆盖以及更好的性能
    更多编译时 （compile-time）提醒以减少 runtime 开销
    virtual DOM 完全重写，mounting & patching 提速 100%
2.更小
    Tree-shaking 更友好
    新的 core runtime：~ 10kb gzipped
```

vue2 响应式的缺点是

```
无法监听到对象属性的动态添加和删除
无法监听到数组下班和length长度的变化
```

computed计算属性和watch区别

```
computed 和 watch 都是vue框架中的用于监听数据变化的属性

功能：computed是计算属性；watch是监听一个值的变化执行对应的回调
是否调用缓存：computed函数所依赖的属性不变的时候会调用缓存；watch每次监听的值发生变化时候都会调用回调
是否调用return：computed必须return；watch可以没有
是否支持异步：computed函数不能有异步；watch可以
使用场景：computed当一个属性受多个属性影响的时候；例如购物车商品结算；watch当一条数据影响多条数据的时候，例如搜索框
```

```
computed：
    在调用的时候不用加（）
    通过属性计算而得来的属性
    对于任何复杂逻辑或一个数据属性在它所依赖的属性发生变化时，也要发生变化，这种情况下，我们最好使用计算属性computed
    属性的结果会被缓存，除非依赖的响应式属性变化才会重新计算。主要当作属性来使用
    如果函数所依赖的属性没有发生变化，从缓存中读取
    当中的函数必须用return返回最终的结果

watch：
    不需要调用
    属性监听，监听属性的变化
    在数据变化时执行异步或开销较大的操作时使用
    watch 一个对象，键是需要观察的表达式，值是对应回调函数。主要用来监听某些特定数据的变化，从而进行某些具体的业务逻辑操作

    watch中的函数有两个参数，前者是newVal，后者是oldVal
    watch只会监听数据的值是否发生改变，而不会去监听数据的地址是否发生改变，要深度监听需要配合deep：true属性使用
```

```
使用场景：
    computed：
        当一个属性受多个属性影响的时候就需要用到computed
        最典型的例子： 购物车商品结算的时候
    watch：
        当一条数据影响多条数据的时候就需要用watch
        搜索数据
```

$set的用法

给data对象新增属性，如果直接赋值操作，虽然可以新增属性，但是不会触发视图更新，要处理这种情况，我们可以使用$set()方法，既可以新增属性,又可以触发视图更新。

用来更新数组或对象，参数1:参数是需要更新的数组或对象，参数2:是数组的下标或者对象的属性名，参数3:是更新的内容

如果是一个对象，我们可以给他动态增添一些属性，并且保证这些属性是个响应式的!

react 有什么特性

```
JSX语法，之前jsx是react独有的，vue中也使用jsx
单向数据绑定
虚拟DOM
声明式编程（不需要关系底层，按照框架规则写即可）
Component(组件化系统)
借助这些特性，react整体使用起来更加简单高效，组件式开发提高了代码的复用率
```

vue和react的区别

1.react是基于mvc ，vue是基于mvvm
2. vue实现了数据的双向绑定，react数据流动是单向的
    vue通过v-model实现数据双向绑定，是v-on：input和v-bind：value的语法糖，双向数据绑定是通过数据劫持 结合 发布订阅模式的方式， 通过Object.defineProperty()来劫持各个属性的setter,getter,在数据变动时发布消息给订阅者，触发相应的监听回调。
    react单向数据流，规范了数据的流向，规定数据由外层组件向内层组件进行，要一级一级的传递，不能反向传递，因为如果子组件可以反向改变父组件的数据，会导致其他子组件数据紊乱，会造成不可控的操作
3.底层原理不同，react有自己的实现方式 ，vue2.0使用object.defindProperty实现的，3.0使用proxy
4.react使用 PureComponent 和 shouldComponentUpdate 优化组件，在Vue 组件的依赖是在渲染过程中自动追踪的，
Vue 通过 getter/setter 以及一些函数的劫持，能精确知道数据变化，不需要特别的优化就能达到很好的性能
5.在 Vue 中我们组合不同功能的方式是通过 mixin，而在React中我们通过 HoC (高阶组件）
6.在 React 中，所有的组件的渲染功能都依靠 JSX，Vue 默认推荐的还是模板，也提供了渲染函数，同时也支持 JSX
7.Vue就相对来说更小更灵活，适合用来开发单页面程序这样一个简单化的组件，React更适合数据经常变化以及构建大型项目的复杂组件

函数组件提供的方法和作用

1.React.createElement： 创建虚拟dom
2.React.createRef： 创建ref对象
3.React.Children: 提供操作对象和数组的方法
4.React.createContext： 创建上下文环境
5.React.cloneElement： 克隆元素或组件
6.React.memo: 优化函数组件

局部组件

全局组件和局部组件的区别

全局组件需要注册，局部组件需要挂载

局部组件中data为什么必须是函数

1.如果data是对象的话，会因为不改变地址发生命名污染问题（同地址对象，如果键值相同，相关内容会互相影响）
2.如果data是函数，函数私有作用域，返回一个新对象

vue的基础指令及作用

v-html：绑定数据到元素的innerHTML上；
v-text：和v-html类似，区别是不识别标签，和{{}}一样；
v-show：通过css的display来显示和隐藏元素；
v-if：通过v-if的布尔值来控制元素在dom树上的挂载和卸载；
v-for：循环遍历元素，循环的载体和后代的元素都会遍历；
v-on：绑定事件 @ ；
v-bind：数据和元素的属性动态绑定 ： ，
v-model向数据绑定：本身是指令，实际为v-bind:value和v-on:input的语法糖

v-for和v-if的优先级，是否可以绑定在同一个dom标签或vue组件上

v-for优先于v-if
不建议作用于同一元素
原因：v-for先遍历所有的元素，然后渲染所有的元素在视图上，给每一个元素都分别绑定v-if，判断值是否进行元素的卸载和挂载，造成性能的浪费
解决方法：往外包一层父元素，v-if绑定在父元素上

v-for为什么循环列表需要加key值？为什么key值不建议使用索引?

key值作为唯一标识，为了更好的区分各个组件，主要是为了高效的更新虚拟DOM
当以数组为下标的index作为key值时，其中一个元素（例如增删改查）发生了变化就有可能导致所有的元素的key值发生改变

v-if和v-show的区别

v-if 是对条件块的销毁与重建 同时他也是惰性的，初始条件为false时啥也不干直到条件为真 才会加载条件块
V-show 条件块总是会被渲染只是基于css的切换
v-if 有更高的切换开销 v-show则是有更高的初始渲染开销 如果切换比较频繁则使 用v-show比较好

vue-router路由守卫有哪些

拦截路由跳转

全局导航守卫：
    全局前置导航守卫：router.beforeEach((to,from,next)=>{})
    全局后置导航守卫：router.afterEach((to,from)=>{})
路由独享守卫：beforeEnter((to,from,next)=>{})
组件内的导航守卫：
    beforeRouteEnter((to,from,next)=>{})
    beforeRouteLeave((to,from,next)=>{})
    beforeRouteUpdate((to,from,next)=>{})

参数:
    to:将要访问的路由信息对象
    form:将要离开的路由信息对象
    next:放行，允许进入下一个导航对应的组件

vue中的事件修饰符

stop：阻止事件向上级DOM元素传递，使用stop修饰符可以阻止事件继续传播，避免在父级元素上触发事件
once：设置事件只能触发一次，使用once修饰符可以设置只在第一次触发事件的时候触发事件侦听器
prevent：是阻止默认事件的发生，使用prevent修饰符可以设置当链接被单击时阻止页面跳转
self：将事件绑定到自身，只有自身才能触发，使用self修饰符可以设置只监听元素自身而不是它的子元素上触发事件

keep-alive

对页面或者是组件进行缓存保留当前的操作

include:字符串或正则表达式，只有名称匹配的组件会被缓存(缓存包含的组件)
exclude:字符串或正则表达式，任何名称匹配的组件都不会被缓存(不缓存包含的组件)
max 数字，最多可以缓存多少组件实例

钩子函数

activated 缓存的组件激活时调用
deactivated 缓存的组件失活时调用

单向数据流

定义

规范了数据的流向，由外层组件向内层组件进行，要一级一级的传递，不能反向传递

为什么react要使用单向数据流

如果子组件可以反向改变父组件的数据，会导致其他子组件数据紊乱，造成不可控的操作

项目优化

1.减少http请求数量
2.压缩图片、选择合适的图片格式（颜色较多就使用jpg，颜色较少就使用png格式）
3.代码模块化、渲染使用key值
4.路由懒加载
5.打包优化
6.合并静态资源（包括css、js和小图片）
7.css放在页面头部，把js放在页面底部（不会阻塞页面渲染）

setState

setState()修改状态以及触发视图的更新

同步异步

1.setState本身是同步的，react将其设计为异步，在原生事件和异步中，它是同步的
2.在合成事件里，生命周期中，react会将setState处理为异步的表现形式，但不易将setState该为异步方法
3.批量处理时，每次setState不会立即执行，会把它放在队列中等待，执行多次setState改的都是相同属性的话，都只会执行最后一次

为什么将setState设计成异步

每次触发setState都会重新渲染视图，如果是同步，同一时间同一个属性改变，会引发多次render执行，消耗性能，所以设计为异步

！！！！！ url到页面渲染！！！！！

localStorage，sessionStorage，cookie 区别

localStorage和sessionStorage是本地存储，cookie是服务器存储
localStorage是长期存储，只有主动删除才能清空 5M
sessionStorage是临时存储，浏览器关闭之后自动删除 5M
cookie失效时间可以设定，存储和读取都需要服务器环境 4K

1M=1024KB

组成

协议：http https
域名：对应的ip地址，决定一台服务器的位置
端口号：http服务器默认是80，https默认是443，确定当服务器的应用
资源路径：在服务器中查找对应的资源
查询字符串：对请求有更加详细的描述
锚点：只有修改锚点不会刷新页面

http跟https的区别

一个加密一个不加密，https是加密的，加密分为对称加密和非对称加密

对称加密通常使用的是相对较小的密钥，一般小于256 bit。
私钥只能由一方安全保管，不能外泄，而公钥则可以发给任何请求它的人。非对称加密使用这对密钥中的一个进行加密，而解密则需要另一个密钥

对称加密 加密与解密使用的是同样的密钥，所以速度快，但由于需要将密钥在网络传输，所以安全性不高。
非对称加密 使用了一对密钥，公钥与私钥，所以安全性高，但加密与解密速度慢。
解决的办法是将对称加密的密钥使用非对称加密的公钥进行加密，然后发送出去，接收方使用私钥进行解密得到对称加密的密钥，然后双方可以使用对称加密来进行沟通。

1、端口
https的端口是443，http的端口是80，当然两者的连接方式也不太一样
2、传输数据
HTTP是超文本传输协议，信息是明文传输，HTTPS则是具有安全性的SSL加密传输协议。https具有安全性
3、申请证书
https传输一般是需要申请证书，申请证书可能会需要一定的费用，而http不需要
4、HTTP的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比HTTP协议安全。

http1.0和http1.1区别

1.连接方面：http1.0 默认使用短连接，每次请求都需要建立新的TCP连接，连接不能复用，http1.1 默认使用长连接，来使多个http请求复用同一个TCP连接，以此来避免使用长连接时每次需要建立连接的消耗和延迟，提高效率

2.资源请求方面：在 http1.0 中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，http1.1 则在请求头引入了 range 头域，它允许只请求资源的某个部分，即返回码是 206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。

3.缓存方面：在 http1.0 中主要使用 header 里的 If-Modified-Since,Expires来做为缓存判断的标准，http1.1则引入了更多的缓存控制策略例如 Etag、If-Unmodified-Since、If-Match、If-None-Match 等更多可供选择的缓存头来控制缓存策略。

4.http1.1 中新增了 host 字段，用来指定服务器的域名。http1.0 中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的 URL 并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上可以存在多个虚拟主机，并且它们共享一个IP地址。因此有了 host 字段，就可以将请求发往同一台服务器上的不同网站。

5.http1.1 相对于 http1.0 新增了很多方法，如 PUT、HEAD、OPTIONS 等

hash和history的区别

1.hash中有#号，history中没有#号
2.hash模式原理是hashchange事件，history模式用h5 history对象原理
3.hash模式不会向服务器发起请求，history模式会向服务器发起请求，需要服务器做配置
4.history模式对老版本有兼容问题

get和post请求的区别

1.post安全性更高，因为get方法发送的数据不受保护，数据在URL栏中是公开的，post方法发送的数据是安全的，因为数据未在URL栏中公开
2.post传递的数量更大，get数据是在URL中发送的，post数据是在正文主体中发送的
3.get方法的数据是可缓存的，而post方法的数据是无法缓存的
4.get方法主要用于获取信息。而post方法主要用于更新数据
5.get的执行率更高

TCP和UDP

TCP和UDP都是工作在传输层的,该层次的目的就是为了实现端到端之间的传输
TCP实现基于连接的可靠的数据通讯。
UDP提供基于无连接的尽力而为的通讯服务。
区别：
　　我们生活中常见的通讯方式大概可以分为两种：打电话和发短信，短信的传输在意的是对方是否收到，收到的信息是否完整，电话的通讯更加在意电话是否拨通，通讯是否紊乱
　　短信的传输更像是UDP进行传输：只管将数据发送出去，但是不确定对象是否能接受或者已经接受，UDP尽自己的能力去发送，实现无连接的数据传输
　　电话的传输就像是使用TCP进行传输：每一次数据的发送都需要得到对应的回应，数据是否接受、接受状态如何、数据顺序是否出错等等，在建立连接的基础上实现传输。

TCP的传输过程

TCP是基于连接进行数据传输，而TCP的连接过程一共有以下三步：三次握手、传输数据、四次挥手。在建立连接的时候，客户端和主机之间会来回发送数据三次，所以建立连接的过程成为三次握手，建立连接之后才能开始正确的传输数据。

在传输完数据之后，两端需要断开连接，断开连接的过程会使两端设备实现四次数据交互，所以拆散连接的过程被称为四次挥手。

UDP的传输过程

UDP协议的处理数据包的过程非常简单，在原有数据包的基础上，加上一些带有标识性的信息段，再通过网卡传输，之后不再关注该数据包的状态，从本质来说，这些数据包之间并没有实质上的联系，所以UDP的数据包传输对于设备的计算消耗来说相当小，可以大大的提高数据传输的传输速率。
简单来说就是，将需要寄出的东西装好，写上一些快递信息，之后交给快递小哥发送快递，我就不需要管了

UDP的传输速率高于TCP，TCP可靠性高于UDP

长连接和短连接

长连接：长连接意味着进行一次数据传输后，不关闭连接，长期保持连通状态。如果两个应用程序之间有新的数据需要传输，则直接复用这个连接，无需再建立一个新的连接。

优点：在多次通信中可以省去连接建立和关闭连接的开销

缺点：需要花费额外的精力来保持这个连接一直是可用的

短连接：短连接意味着每一次的数据传输都需要建立一个新的连接，用完再马上关闭它。下次再用的时候重新建立一个新的连接，如此反复。

优点：由于每次使用的连接都是新建的，所以基本上只要能够建立连接，数据就大概率能送达到对方

缺点：每个连接都需要经过三次握手和四次握手的过程，耗时大大增加

this.$nestTick

this.$nestTick将回调延迟到下次 DOM 更新循环之后执行。

componentDidCatch

如果 render() 函数抛出错误，该函数可以捕捉到错误信息，并且可以展示相应的错误信息。

页面加载过程

1.向浏览器输入网址
2.浏览器根据 DNS 服务器得到域名的 IP 地址
3.向这个 IP 的机器发送 HTTP 请求
4.服务器收到、处理并返回 HTTP 请求
5.浏览器接收到服务器返回的内容

浏览器渲染原理

1.浏览器首先使用HTTP协议或者HTTPS协议，向服务器端请求页面
2.把请求回来的HTML代码经过解析，构建成DOM树
3.计算DOM树上的CSS属性
4.根据CSS属性对元素逐个进行渲染
5.一个可选的步骤是对位图进行合成，这会极大地增加后续绘制的速度
6.合成之后，在绘制到界面上

！！！！！JS！！！！！

闭包

函数嵌套函数，里部变量能访问外部变量，这个变量称为自由变量，这个变量的集合称为闭包
闭包会造成内存泄露

作用

1.延伸变量的作用范围
2.变量不会销毁（核心作用）
3.变量什么时候不会销毁：变量被引用并且有内存

闭包形成的条件

1.函数嵌套
2.将内部函数作为返回值返回
3.内部函数必须使用到外部的变量

闭包的使用场景

1. 使用闭包代替全局变量；
2. 函数外或在其他函数中访问某一函数内部的参数；
3. 在函数执行之前为要执行的函数提供具体参数；
4. 在函数执行之前为函数提供只有在函数执行或引用时才能知道的具体参数；
5. 暂停执行；
6. 包装相关功能。

闭包的缺点

内存泄漏（栈溢出）

垃圾回收机制

垃圾回收机制的原理是找到不再继续使用的变量，释放其内存。垃圾回收器会按照固定的时间间隔(或代码中预定的收集时间)，周期性地执行这一操作

Javascript 会找出不再使用的变量，不再使用意味着这个变量生命周期的结束。Javascript 中存在两种变量——全局变量和局部变量，全部变量的声明周期会一直持续，直到页面卸载

而局部变量声明在函数中，它的声明周期从执行函数开始，直到函数执行结束。在这个过程中，局部变量会在堆或栈上被分配相应的空间以存储它们的值，函数执行结束，这些局部变量也不再被使用，它们所占用的空间也就被释放

垃圾回收的两种实现方式：标记清除、引用计数

标记清除

主要思想是给当前不使用的值加上标记，然后再回收他的内存

垃圾回收器在运行时会给存储在内存中的变量加上标记，然后他会去掉环境变量和被环境变量引用的变量的标记，此后被加上标记的变量(环境变量中没有使用访问的变量)就是准备删除的变量;最后垃圾回收器完成清除工作，销毁那些带标记的值，并回收他们占用的内存空间

过程：根节点（一般来说，根是代码中引用的全局变量）=> 算法检查所有根节点和他们的子节点并且把他们标记为活跃的(意思是他们不是垃圾)。任何根节点不能访问的变量将被标记为垃圾 => 垃圾收集器释放所有未被标记为活跃的内存块，并将这些内存返回给操作系统

引用计数

跟踪记录所有值被引用的次数

例如变量a赋值后，这个值的引用次数为1，这个a值又被赋值给另一个变量b，这时引用次数+1;但当b赋另外的值，引用次数-1;当值的引用书为0，说明没有办法再访问这个值，这时就可以将内存回收了。

IE9以下还在使用引用计数，当对象循环引用时，引用次数无法标记为0，就会导致无法被回收。其他浏览器废弃使用

es6新增

1.用const和let声明变量
2.新增了数组方法和对象方法
3.新增了解构赋值（数组的解构赋值，精髓在于下标的一一对应）
4.新增了模板字符串和箭头函数

let，const和var的区别

let:声明不能提前,不能重复声明,块级作用域,暂时性死区,声明时可以不用赋值,可以修改值
const:声明不能提前,不能重复声明,块级作用域,暂时性死区,一旦声明必须赋值，声明的变量值不能修改
var:声明提前,能重复声明但以前的值会被覆盖,声明时可以不用赋值,可以修改值

es6新增的数组方法

arr.forEach() 接收一个函数，函数依次执行，执行次数是数组长度，每次执行都接收三个参数：数组元素，元素索引，数组本身

arr.map() 接收一个函数，函数依次执行，执行次数是数组长度，返回一个数组，数组每一项是回调函数的返回值

arr.some() 判断数组中是否有一项符合条件，只要有一个符合，返回true，如果所有元素都不符合，返回false

arr.every() 判断数组所有的项是不是都符合条件，所有的都符合，返回true,只要有一个不符合，返回false

arr.find() 查找数组的某一项 找到之后返回该元素 找不到返回undefined 只找一个

arr.findIndex() 查找数组的某一项 找到之后返回该元素下标 找不到返回-1 只找一个

arr.filter() 返回符合条件的元素组成的新数组

arr.reduce() 第一个参数：上一次执行的返回值,第二个参数：next我们下一个要遍历的元素;第三个参数：index(第二个参数的下标);第四个参数：array(原数组)

forEach和map的区别

map方法的参数是一个回调函数，回调函数三个参数分别是遍历数组的每一项,当前项的下标,原数组，map方法循环次数为数组length长度，根据回调函数的返回值，创建一个新数组，返回值是新数组

foreach方法的参数是一个回调函数，回调函数三个参数分别是遍历数组的每一项,当前项的下标,原数组，foreach方法循环次数为数组length长度，每次循环执行回调函数内的代码

相同点

1.都是循环遍历数组中的每一项。
2.每次执行匿名函数都支持三个参数，参数分别为item(当前每一项)，index(索引值)，arr(原数组)。
3.匿名函数中的this都是指向window。
4.只能遍历数组。

不同点

map()会分配内存空间存储新数组并返回，forEach()不会返回数据。
forEach()允许callback更改原始数组的元素。map()返回新的数组。

es6新增的对象方法

Object.assign() 合并对象

Object.keys() 遍历对象，返回该对象key组成的数组

Object.is() 判断两个值是否是相同的值，跟比较运算符===作用一样，仅仅修复了NaN与NaN的行为

Object.values() 遍历对象，返回该对象value组成的数组

super

Super指向当前函数所挂靠实例所属类的原型
class在继承后使用this需要super关键字

super作为函数在子类的constructor中调用时，代表的是父类的构造函数
super作为对象：
    super用在实例方法中：
        1、super指向父类的原型对象
        2、通过super调用父类方法时，super内部的this指向子类的实例
        3、当通过super为子类属性赋值时，super就是this
    在子类的静态方法中通过super调用父类方法时，super内部的this指向子类（不是子类的实例）

面向对象

1.面向对象的含义：通过函数封装得到的有一个类（函数）
2.每个类(函数)天生有一个prototype的属性，这个prototype又是一个对象，这个对象里有个constructor（构造函数）的属性，属性值是类本身。
3.我们所有new 一个类的时候，其实是调用它的构造函数。构造函数里的属性都是私有的，构造函数里的this都是实例对象。
4.每个对象天生有一个__proto__,指向类的原型。
5.Prototype和__proto__prototype是类的或者函数的，__proto__是对象的Prototype是存储机制，程序员来实现。__proto__是查找机制（浏览器的）
6.原型链首先定义一个对象，其次看这个对象的属性是否是私有的，是就直接使用，不是的就通过__proto__往他的类的prototype上查找，有就直接使用，没有就继续向上查找，直到查找到基类Object，没有就是undefined，有就直接使用。这种查找机制 叫原型链。

万物皆对象

封装 继承 多态

宏任务和微任务

宏任务：ajax，setTimeout，setInterval，
微任务：Promsie.then，.catch，.finally，process.nextTick(()=>{})
主线程 -》 微任务 -》 宏任务

深拷贝和浅拷贝

区别：浅拷贝只复制对象的第一层属性，而深拷贝会对对象的属性进行递归复制。

浅拷贝

会创建一个新对象，对原有对象的成员进行依次拷贝。如果属性是基本类型，拷贝的就是基本类型的值；如果属性是引用类型，拷贝的就是内存地址。因此如果新对象中的某个对象成员改变了地址，就会影响到原有的对象。

1.赋值
2.Object.assign():该方法可以实现浅拷贝，也可以实现一维对象的深拷贝。

该方法需要注意的是：
如果目标对象和源对象有同名属性，或者多个源对象有同名属性，则后面的属性会覆盖前面的属性。

如果该函数只有一个参数，当参数为对象时，直接返回该对象；当参数不是对象时，会先将参数转为对象然后返回。

因为null 和 undefined 不能转化为对象，所以第一个参数不能为null或 undefined，会报错。

3、展开运算符:使用展开运算符可以在对对象或数组的第二层开始的拷贝是浅拷贝。

4、数组方法实现数组浅拷贝

（1）Array.slice

array.slice(start, end)，该方法不会改变原始数组

该方法有两个参数，两个参数都可选，如果两个参数都不写，就可以实现一个数组的浅拷贝。

（2）Array.concat

用于合并两个或多个数组。此方法不会更改现有数组，而是返回一个新数组。该方法有两个参数，两个参数都可选，如果两个参数都不写，就可以实现一个数组的浅拷贝。

深拷贝

(1)JSON.parse(JSON.stringify())：这种方法可以实现数组和对象和基本数据类型的深拷贝，但不能处理函数、正则。

(2)递归：

```
function deepCopy(obj, parent = null) {
let result
let _parent = parent
while(_parent) {
if (_parent.originalParent === obj) {
return _parent.currentParent
}
_parent = _parent.parent
}
if (obj && typeof(obj) === 'object') {
if (obj instanceof RegExp) {
result = new RegExp(obj.source, obj.flags)
} else if (obj instanceof Date) {
result = new Date(obj.getTime())
} else {
if (obj instanceof Array) {
    result = []
} else {
    let proto = Object.getPrototypeOf(obj)
    result = Object.create(proto)
}
for (let i in obj) {
    if(obj[i] && typeof(obj[i]) === 'object') {
    result[i] = deepCopy(obj[i], {
    originalParent: obj,
    currentParent: result,
    parent: parent
    })
    } else {
    result[i] = obj[i]
    }
}
```

```
    }
    } else {
    return obj
    }
    return result
    }
```

数组拍平

(1)递归方法

```
    var arr = [ [1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14] ] ]
], 10];
    function fn(arr){
        let arr1 = []
        arr.forEach((val)=>{
            if(val instanceof Array){
                arr1 = arr1.concat(fn(val))
            }else{
                arr1.push(val)
            }
        })
        return arr1
    }
    console.log(fn(arr))
```

(2)reduce实现

```
    var arr = [ [1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14] ] ]
], 10];
    function fn(arr){
        return arr.reduce((prev,cur)=>{
            return prev.concat(Array.isArray(cur)?fn(cur):cur)
        },[])
    }
    console.log(fn(arr))
```

(3)toString方法

```
    var arr = [ [1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14] ] ]
], 10];
    arr =  arr.toString().split(',')
    console.log(arr)
```

(4)扩展运算符

```
    var arr = [ [1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14] ] ]
], 10]
    function fn(arr){
        let arr1 = [];
        let bStop = true;
        arr.forEach((val)=>{
            if(Array.isArray(val)){
                arr1.push(...val);
```

```
                bStop = false
            }else{
                arr1.push(val)
            }
        })
        if(bStop){
            return arr1;
        }
        return fn(arr1)
    }
    console.log(fn(arr))
```

箭头函数和普通函数的区别

1、外形不同：箭头函数使用箭头定义，普通函数中没有。
2、 箭头函数全都是匿名函数：普通函数可以有匿名函数，也可以有具名函数
3、箭头函数不能用于构造函数：普通函数可以用于构造函数，以此创建对象实例。
4、 this 的指向不同：在普通函数中，this 总是指向调用它的对象，如果用作构造函数，它指向创建的对象实例。箭头函数的this指向的是在你书写代码时候的上下文环境对象的this，如果没有上下文环境对象，那么就指向最外层对象window。
5、箭头函数不具有 arguments 对象：每一个普通函数调用后都具有一个arguments 对象，用来存储实际传递的参数。但是箭头函数并没有此对象。（箭头函数的arguments指向其父级函数作用域的arguments）
6、普通函数存在变量提升现象，箭头函数不可以，（会报错）

7、箭头函数不具有 prototype 原型对象。箭头函数不具有 super。箭头函数不具有 new.target（new.target用来检测函数是否被当做构造函数使用，他会返回一个指向构造函数的引用。因为箭头函数不能当做构造函数使用，自然是没有new.target的）

事件捕获和事件冒泡的区别

标准事件流三个阶段

捕获阶段、目标阶段、冒泡阶段
事件捕获：从最外面开始触发的，直到捕获到你点击的div为止
事件冒泡：事件的向上传导，里到外
目标阶段：目标元素触发

取消默认行为

e.preventDefault()
return false 也可以取消默认行为，但是不同，主要是用来返回值的，只是顺带取消默认行为

阻止冒泡

e.stopPropagation()
e.cancelBubble = true 也可以阻止冒泡，用来兼容老版本浏览器（IE）（不符合W3C标准，只有IE识别，但是新版本的Chrome、Opera等是识别的）

状态码

100-199 用于指定客户端相应的某些动作
200-299 用于表示请求成功
300-399 用于已经移动的文件并且常被包含在定位头信息中指定新的地址信息
400-499 用于指出客户端的错误
500-599 用于指出服务器的错误

200：（成功）表示请求成功，服务器已成功处理了请求。一般用于相应GET和POST请求。
201：（已创建）请求成功并且服务器创建了新的资源。
202：（已接受）服务器已接受请求，但尚未处理。
203：（非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。
204：（无内容） 服务器成功处理了请求，但没有返回任何内容。
205：（重置内容） 虽然没有新文档但浏览器要重置文档显示。这个状态码用于强迫浏览器清除表单域。
206：（部分内容） 服务器成功处理了部分 GET 请求。

300：（多种选择） 针对请求，服务器可执行多种操作。 服务器可根据请求者 (user agent) 选择一项操作，或提供操作列表供请求者选择。
301：（永久重定向）请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。
302：（暂时重定向）服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。
303：（查看其他位置） 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码。
304：（未修改） 自从上次请求后，请求的网页未修改过。 服务器返回此响应时，不会返回网页内容。
305：（使用代理） 请求者只能使用代理访问请求的网页。 如果服务器返回此响应，还表示请求者应使用代理。
307：（临时重定向） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。

400：（错误请求） 服务器不理解请求的语法。
401：（未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。
403：（禁止） 服务器拒绝请求。
404：（未找到） 服务器找不到请求的网页。
405：（方法禁用） 禁用请求中指定的方法。
406：（不接受） 无法使用请求的内容特性响应请求的网页。
407：（需要代理授权） 此状态代码与 401（未授权）类似，但指定请求者应当授权使用代理。
408：（请求超时） 服务器等候请求时发生超时。
409：（冲突） 服务器在完成请求时发生冲突。 服务器必须在响应中包含有关冲突的信息。
410：（已删除） 如果请求的资源已永久删除，服务器就会返回此响应。
411：（需要有效长度） 服务器不接受不含有效内容长度标头字段的请求。
412：（未满足前提条件） 服务器未满足请求者在请求中设置的其中一个前提条件。
413：（请求实体过大） 服务器无法处理请求，因为请求实体过大，超出服务器的处理能力。
414：（请求的 URI 过长） 请求的 URI（通常为网址）过长，服务器无法处理。
415：（不支持的媒体类型） 请求的格式不受请求页面的支持。
416：（请求范围不符合要求） 如果页面无法提供请求的范围，则服务器会返回此状态代码。
417：（未满足期望值） 服务器未满足"期望"请求标头字段的要求。

500：服务器内部错误 服务器遇到错误，无法完成请求。
501：（尚未实施） 服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码。
502：（错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。
503：（服务不可用） 服务器目前无法使用（由于超载或停机维护）。 通常，这只是暂时状态。
504：（网关超时） 服务器作为网关或代理，但是没有及时从上游服务器收到请求。
505：（HTTP 版本不受支持） 服务器不支持请求中所用的 HTTP协议版本。

fs模块

读取文件：fs.readFileSync()
写入文件：fs.writeFileSync()
删除文件：fs.unlinkSync()
创建目录：fs.mkdirSync()
读取目录：fs.readdirSync()
删除目录：fs.rmdirSync()
判断是否是文件夹或文件：fs.statSync().isDirectory() fs.statSync().isFile()
判断文件是否存在：fs.existsSync()

！！！！！ HTML！！！！！

HTML5

优点

1.有利于维护以及协同开发
2.提高浏览器渲染解析速度
3.有利于搜索引擎的爬取信息

h5新增标签

1.header 头部标签 2.footer 尾部标签 3.main 主体标签 4.nav 导航标签
5.aside 侧边栏标签 6.article 文章标签 7.section 段落以及楼层

h5新增特性

1.新增语义化标签
    1.header：头部标签
    2.footer：尾部标签
    3.main：主题标签
    4.nav：导航标签
    5.aside：侧边栏标签
    6.article：文章标签
    7.section：段落及楼层
2.视频和音频标签
     <video src="url" controls="controls"></video>
     <audio src="url" loop="loop" muted="muted"></audio>
3.绘图
    <canvas id="myCanvas" width="200" height="100"></canvas>
4.Form 表单增强
     input 的type 新属性: number, data, month, week, time, email, url 等

真实DOM和虚拟DOM

什么是虚拟DOM

虚拟 dom 是用 js 对象来表示真实的 DOM 树结构，创建一个虚拟 DOM 对象

为什么需要虚拟DOM

渲染视图会调用render函数，这种渲染不仅发生在组件创建时，同时发生在视图依赖的数据更新时。如果在渲染时，直接使用真实的DOM，由于真实DOM的创建，更新，插入等操作会带来大量的性能损耗，从而就会极大的降低渲染效果。使用虚拟DOM来代替真实DOM，主要为解决渲染效率问题

虚拟DOM如何转换为真实DOM

在一个组件实例首次被渲染时，它先生成虚拟DOM树，然后根据虚拟DOM树创建真实DOM，并把真实DOM挂载到页面中合适的位置，此时，每个虚拟DOM便会对应一个真实DOM。如果一个组件受响应式数据变化的影响，需要重新渲染时，它仍然会重新调用render函数，创建出一个新的虚拟DOM树，用新树和旧树对比，通过对比，vue会找到最小更新量，然后更新必要的虚拟DOM节点，最后，这些更新后的虚拟节点，会去修改他们对应的真实DOM，这样一来，就保证了对真实DOM达到最小改动。

虚拟DOM和真是DOM的区别

真实DOM: 文档对象模型，是一个结构化文本的抽象，在页面渲染出的每一个结点都是一个真实DOM结构
虚拟DOM:本质上是以 JavaScript 对象形式存在的对 DOM 的描述，创建虚拟DOM目的就是为了更好将虚拟的节点渲染到页面视图中，虚拟DOM对象的节点与真实DOM的属性一一照应

区别:
1.虚拟DOM不会进行排版与重绘操作，而真实DOM会频繁重排与重绘
2.虚拟DOM的总损耗是"虚拟DOM增删改+真实DOM差异增删改+排版与重绘"，真实DOM的总损耗是"真实DOM完全增删改+排版与重绘"
3.真实DOM:浏览器中提供的概念，用JS对象，表示页面上的元素，并提供了操作DOM对象的API
4.虚拟DOM:框架中的概念，需要手动用JS对象来模拟DOM元素和嵌套关系

虚拟DOM和真实DOM的优缺点

真实DOM的优势:易用
真实DOM的缺点:
1.效率低，解析速度慢，内存占用量过高
2.性能差:频繁操作真实DOM，易于导致重绘与回流
虚拟DOM的优势:
简单方便:如果使用手动操作真实DOM来完成页面，繁琐又容易出错，在大规模应用下维护起来也很麻烦
性能方面:使用Virtual DOM，能够有效避免真实DOM数频繁更新，减少多次引起重绘与回流，提高性能
跨平台:React借助虚拟DOM， 带来了跨平台的能力，一套代码多端运行
虚拟DOM的缺点:
在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化
首次渲染大量DOM时，由于多了一层虚拟DOM的计算，速度比正常稍慢

创建一个虚拟DOM

```js
<script type="text/babel">
    const Vspan =
        React.createElement('span', { id: "box", class: 'box' }, 'helle World')
    ReactDOM.render(Vspan, document.querySelector('#test'))
</script>
```

script中defer和async两个属性的区别

defer是等待执行，async是立即执行

块标签和行内标签的区别

块标签：
    1.块标签始终独占一行
    2.默认宽高是父级的宽高
    3.可以直接设置宽高
    div，p，ul，ol，li，
行内标签：
    1.多个行内标签可以呈一行显示
    2.默认宽高是标签内容的宽高
    3.不能直接设置宽高
    span，a,i,em
行内块标签：
    1.多个行内块标签可以呈一行显示
    2.默认宽高是标签内容的宽高
    2.可以直接设置宽高
    img，textarea，input

嵌套规则：
    1.行标签不能嵌套块标签
    2.块标签可以嵌套任何标签
    3.一般情况下我们尽量保持同一级别的标签在输出特征上保持一致

！！！！！ CSS！！！！！

C3新增特性

1.CSS3选择器
    1.基本选择器，2.属性选择器，3.伪类选择器
2.CSS3边框与圆角
    1.CSS3圆角border-radius，2.盒阴影box-shadow
3.CSS3背景与渐变
    1.CSS3背景：background-image，2.CSS3渐变：可以在两个或者多个指定颜色之间显示平移的过渡
4.CSS3过渡
    允许css的属性值在一定时间区间内平滑的过渡，在鼠标点击，鼠标滑过或对元素任何改变中触发，并圆滑地以动画形式改变css的属性值。
5.CSS3变换
    让一个元素在一个坐标系统中变形，这个属性包含一系列的变形函数，可以移动，旋转，缩放元素。
6.CSS3动画
    使元素从一种样式逐渐变化到另外一种样式的效果。

rem和em的区别

rem是根据根元素的font-size大小来决定的
em是根据自身或者父元素的font-size大小来决定的

标准盒模型和怪异盒模型（IE盒模型）

标准盒模型：box-sizing:content-box
怪异盒模型：box-sizing:border-box

区别：
　　标准盒模型：元素宽高包含内容宽高，不包含padding和border，添加padding和border之后元素宽高向外扩张
　　怪异盒模型：元素宽高包含内容宽高，以及padding和border，添加padding和border之后元素宽向内挤压

清除浮动的方法

```
1.给浮动元素的直接父级设置overflow：hidden
2.给浮动元素的直接父级设置伪类：
    .clearfix:after{
        content:''; //设置内容为空
        display:block; //空的块标签
        visbility:hidden; //这个空标签不可见
        clear:both; //清除浮动
    }
```

flex

容器属性

```
1.flex-direction:控制容器的主轴方向
    row:从左往右的水平主轴
    column:从上往下的垂直主轴
    row-reverse:从右往左的水平主轴
    column-reverse:从下往上的垂直主轴

2.flex-wrap:控制项目的换行方式
    nowrap:不换行
    wrap:换行
    wrap-reverse:反向换行

3.flex-flow:flex-direction和flex-wrap的复合属性

4.justify-content:控制容器在主轴方向的对齐方式
    flex-start:起点对齐
    flex-end:终点对齐
    center:中心点对齐
    space-between:两端对齐
    space-around:等边距对齐

5.align-items:控制容器在交叉轴方向的对齐方式

    flex-start:起点对齐
    flex-end:终点对齐
    center:中心点对齐
    baseline:第一行文字基线对齐
    stretch:没有设置高度时占满整个交叉轴位置

6.align-content:控制多条主轴的对齐方式
```

```
    flex-start:起点对齐
    flex-end:终点对齐
    center:中心点对齐
    space-between:两端对齐
    space-around:等边距对齐
    stretch:轴线占满整个交叉轴
```

让元素水平垂直居中的方法

```
1.flex布局：display: flex;align-items: center;justify-content：center；
2.定位：父元素position: relative，子元素position: absolute
3.padding和margin
4.line-height,text-align
5.父元素 display: flex，子元素margin: auto
6.定位+translate：父元素position: relative，子元素position: absolute;top:
50%;transform: translate(-50%, -50%)
7.left: calc(50% - 50px);top: calc(50% - 50px);
8.grid布局 display: grid; align-items:center; justify-content:center;
```

display:none 和 visibility: hidden 和 opacity: 0 的区别

```
display:none: 会让元素完全从渲染树中消失，渲染的时候不占据任何空间，不能点击，
visibility: hidden:不会让元素从渲染树消失，渲染元素继续占据空间，只是内容不可见，不能点击
opacity: 0: 不会让元素从渲染树消失，渲染元素继续占据空间，只是内容不可见，可以点击

继承：
display: none和opacity: 0：是非继承属性，子孙节点消失由于元素从渲染树消失造成，通过修改子孙
节点属性无法显示。
visibility: hidden：是继承属性，子孙节点消失由于继承了hidden，通过设置visibility:
visible;可以让子孙节点显式。

性能：
displaynone : 修改元素会造成文档回流,读屏器不会读取display: none元素内容，性能消耗较大
visibility:hidden: 修改元素只会造成本元素的重绘,性能消耗较少读屏器读取visibility: hidden
元素内容
opacity: 0 ： 修改元素会造成重绘，性能消耗较少，opacity 使用场景： 自定义图片上传按钮
```

文本溢出显示省略号

单行文本溢出显示省略号

```
text-overflow:ellipsis; 超出显示省略号
white-space:nowrap; 强制不换行
overflow:hidden; 溢出隐藏
```

多行文本溢出显示省略号

```
display:-webkit-box; 弹性盒处理
-webkit-box-orient:vertical; 竖向排列
-webkit-line-clamp:3;控制显示行数
overflow:hidden; 溢出隐藏
```

瀑布流

父级盒子：
    column-count:2; 一行显示几个元素
    column-gap：5px; 列间距
子级盒子：
    break-inside:avoid; 控制文本块分解成单独的列，以免项目列表的内容跨列，破坏整体布局

multi-column实现瀑布流

multi-column实现瀑布流主要依赖以下几个属性：

column-count: 设置共有几列
column-width: 设置每列宽度，列数由总宽度与每列宽度计算得出
column-gap: 设置列与列之间的间距
column-count和column-width都可以用来定义分栏的数目，而且并没有明确的优先级之分。优先级的计算取决与具体的场景。

计算方式为：计算column-count和column-width转换后具体的列数，哪个小就用哪个。

flexbox实现瀑布流

html结构：

```
<div class="masonry">
    <!-- 第一列 -->
    <div class="column">
        <div class="item"></div>
        <!-- more items-->
    </div>
    <!-- 第二列 -->
    <div class="column">
        <div class="item"></div>
        <!-- more items-->
    </div>
    <!-- 第三列 -->
    <div class="column">
        <div class="item"></div>
        <!-- more items-->
    </div>
</div>
```

上面代码中div.masonry代表当前瀑布流容器，div.column代表每一列的容器，div.item代表每一列中的每一项。

我们需要将div.masonry和div.column都通过display:flex将其设置为Flex容器。

不同的是瀑布流容器主轴方向设置为水平方向flex-direction:row,列容器主轴方向设置为垂直方向flex-direction:column

```
.masonry {
    display: flex; // 设置为Flex容器
    flex-direction: row; // 主轴方向设置为水平方向
}
```

```
.column {
    display: flex; // 设置为Flex容器
    flex-direction: column; // 主轴方向设置为垂直方向
}
```

由于当前的html结构分为了瀑布流容器和列容器，并且常见的需求图片均是从左至右再从上到下来进行排列，所以需要通过Javascript来区分每一列的具体数据：

假设分为三列，伪代码如下：

```
let data1 = [], //第一列
    data2 = [], //第二列
    data3 = [], //第三列
    i = 0;
while (i < data.length) {
    data1.push(data[i++]);
    if (i < data.length) {
        data2.push(data[i++]);
    }
    if (i < data.length) {
        data3.push(data[i++]);
    }
}
return {
    //第一列
    data1,
    //第二列
    data2,
    //第三列
    data3
};
```

设置文字不被鼠标选中

```
-webkit-user-select:none;
-moz-user-select:none;
-ms-user-select:none;
user-select:none;
```

！！！！！ 简历！！！！！

小程序

生命周期

```
onLoad：页面加载时，一个页面只会调用一次
onShow：页面显示时，每次打开页面都会调用一次
onReady：页面初次渲染完成，一个页面只会调用一次，代表页面已经准备妥当，可以和视图层进行交互
onHide：页面隐藏时，当navigateTo或底部tab切换时调用
onUnload：页面卸载时，当redirectTo或navigateBack的时候调用
```

页面跳转

navigateTo
    最普遍的一种跳转方式，保留当前页面，跳转到应用内的某个页面
    实际应用：小程序中左上角有一个返回箭头，可返回上一个页面。也可以通过方法 wx.navigateBack 返回原页面
redirectTo
    关闭当前页面，跳转到应用内的某个页面
switchTab
    跳转到 tabBar 页面，并关闭其他所有非 tabBar 页面
wx.navigateTo 和 wx.redirectTo 不允许跳转到 tabbar 页面，只能用 wx.switchTab 跳转到 tabbar 页面。当需要自定义tarbar时只能用这个方法跳转
reLaunch
    关闭所有页面，打开到应用内的某个页面
    wx.redirectTo方法不会被加入堆栈,但仍可通过wx.navigateBack(OBJECT)方法返回之前堆栈中的页面
    会清空当前的堆栈

分包

分包：指的是把一个完整的小程序项目,按照需求划分为不同的子包,构建的时候打包成不同的分包,按需加载

好处：可以减少小程序首次启动的下载时间　　可以多人开发中更好的解耦协作

分包体积的限制：整个小程序,主包加分包不能超过20M,单个包不能超过2M,主包也是

配置：在 app.json 的 subpackages 节点中声明分包的结构

微信小程序每个分包的大小是2M，总体积一共不能超过20M。如果要分包，那么就需要把分包的模块放在与pages同级，而不能放在下级，放在同级后，要分包也必须得整个模块进行分包。不能同一个模块有的分包，有的不分包，不然微信会报错。首页、公共模块的方法、基础组件等通用模块一定要放在主包，这样在加载分包的时候，保证通用模块存在。最终打包上传的是app.js，在里面可以看见我们分包的实际情况。

鉴权

鉴权是指验证用户是否拥有访问系统的权利。

鉴权就是权限管理，也就是权利限制，不同的权力的人能干不同的事情。他会根据不同权限的人来动态控制页面呈现和按钮的呈现。

登陆的时候获取token值，然后再根据token获取用户信息也就是权限表，再根据权限表添加前端路由，来做到让不同的人看见不同的页面.
我们通过路由来控制不同的页面展示并跳转，每次我们进入另一个页面的时候事实上就是进入不同的路由，这时候我们可以使用路由守卫来进行一些操作限制，我们可以在这里判断cookie是否存有token，如果没有则会进入到登录页进行登录获取token和权限等信息。我们可以根据权限来组装路由表 (关于路由表组装会在下文介绍) 进而通过router.addRoutes添加路由，并把这些信息存储在vuex里面，我们的tabbar侧边栏等可以根据这些信息来控制样式又或者是显示隐藏。
简单来说：
    登录系统获取token
    根据token获取用户信息（权限表等）
    根据权限表等信息动态添加前端路由，让不同的人看见不同的页面

首先我们先在用户登录的时候获取到他的token值，将token值存储到cookie里面，然后根据token获取用户信息，存储用户信息，通过router.addRoutes添加路由,并把这些信息存储在vuex里面,vuex来管理路由，根据路由表来渲染侧边栏或者控制按钮的显示和隐藏

路由鉴权，后台给前端返回的数据，基本上是数组包对象，对象里面包含了：前端路由这个字段，角色信息（管理员，非管理员，普通用户），类型（是否有children，id和pid（父级id））
按钮健全，数组，type(按钮的类型),content（value值），isShow(是否展示)

401 需要身份认证验证
403 禁止访问
    浏览器缓存：协商缓存，强缓存
    强缓存是根据返回头中的Expires 或者 Cache-Control 两个字段来控制的，都是表示资源的缓存有效时间。
    协商缓存是由服务器来确定缓存资源是否可用。
501 服务器不支持的请求方法
304 请求资源与本地缓存相同，未修改
301 永久重定向
302 临时重定向

鉴权的几种方式

1、HTTP Basic Authentication
2、session-cookie
3、Token 验证
4、OAuth(开放授权)

大文件上传

二进制文件

不论是文本文件还是音频视频类的多媒体文件都需要转换成二进制文件，计算机才能识别

MD5验证

MD5是一种被广泛使用的密码散列函数，需要用到文件的MD5值来确保信息传输完整一致，是一种密码框架

为什么使用大文件上传

在同一个请求，要上传大量数据，导致接口请求的时间很漫长，或许会造成接口超时的后果，且上传过程中如果出现网络异常，那么重新上传还是从零开始上传，大文件上传可以完美解决了以上的弊端，且支持暂停和继续的功能。

分片上传

首先我们先要获取到上传的这个数据，先将上传的图片或者视频解析为BUFFER数据，然后将它转换为二进制文件，然后我们会把文件切片处理，就是把一个文件分割成好几个部分，可以是固定数量或者固定大小，每一个切片都有自己的部分数据和自己的名字，flie中有一个slice方法，通过这个方法，我们就可以对二进制文件进行拆分，我们需要拿到原文件的hash值，因为hash 值是不会变的，这样我们就可以避免文件改名后重复上传的问题

断点续传

当我们上传的过程中出现了网络问题造成强制中断，那么我们要实现当下次上传的时候需要校验一下上传到了哪一步，然后继续上传。所以我们将这段逻辑在服务端完成，前端需要向服务端发送一个请求获取当前已经上传的内容，获取之后需要判断一下是否含有这个文件，如果存在的话我们不需要再进行该切片的上传，这样就实现了断点续传

文件合并

先创建一个数组，将所有的文件切片放在这个数组里，存放时我们可以给切片进行一个排序，这样我们读取目录的时候会更简单清晰，我们传完的切片就可以把它给移除掉。前端发送切片完成后，发送一个合并请求，后端收到请求后，将之前上传的切片文件合并。

流程

大文件上传主要是他的一个切片上传和断点续传，首先我们先来说一下 为什么要使用大文件上传呢，因为比如我们要上传一个视频，他有40分钟或者一个小时，他的数据很大，上传起来很耗费时间，如果传输中断或者是网络异常，就全没了，就要重新开始传，大文件上传就是用到了切片上传跟断点续传来解决这个问题

当我们点击上传的时候，我们获取到这张图片的信息，然后将他解析成buffer数据，再转成二进制文件，这个时候我们会把文件进行切片处理，就是把一个文件分成各个等份，这个等份可以是每个切片的大小或者是可以分为多少份来上传，每一个切片都有自己的数据和名字，我们用到了file中的slice方法来对二进制文件进行拆分，再获取到原文件的hash值，因为hash值是不会变的，这样就可以避免文件改名后重复上传的问题

之后就是一个断点续传，断点续传跟切片上传差不多的，也是将数据分成各个等份，他是有一个暂停和继续的功能的，就是当我们的传输中断，下一次上传的时候需要判断一下上传到哪一步了，然后继续上传就可以了，之后会进行一个文件合并，需要创建一个数组，将所有的文件切片放在这个数组里，我们传完的切片就可以把它给移除掉了。

封装组件

组件可以提升整个项目的开发效率。能够把页面抽象成多个相对独立的模块，解决了我们传统项目开发：效率低、难维护、复用性低等问题。

首先我们要先确定这个组件的用途，在开发项目的时候

有些业务的属性比较复杂，我们需要增添一些自己需要的功能

后台管理系统，我们大多是基于element ui来进行二次封装的，使用slot插槽（vue）高阶组件（react）

有一些属性，插槽是一种方式，也可以通过父传子来实现，可以把属性取出来，整成一个数组，这个数组里面包括这些取出来的属性，在封装的时候，遍历数组里边的东西，调用的时候把数据里边相关的属性传过去（封装的组件里边）就行了

封装地图

1、高德地图实现地图找房功能

实现思路：

我的项目的地图找酒店主要分为三层。第一层为市区层，比如北京市，天津市等；第二层为片区，比如海淀，朝阳等；第三层则为小区。

因为第一层，第二层的数据没有那么多，这两个接口都是把所有的数据一次返回给前端。

但是第三层的数据量就非常的巨大了，我们跟后端小哥商量的是采取的是返回部分数据，将前端页面上显示的最大经纬度以及最小经纬度传给后台，后台再将筛选后的数据返回给前端。

首先需要以异步加载的方式添加高德地图的API。因为项目使用 Vue 进行开发的单页应用，有可能用户并没有进入地图找房的页面，所以项目是在打开地图找酒店的页面时添加高德地图的API。

接下来就是实现自定义覆盖物这个方法了，这个实现主要是参考官方文档。当地图找房对绘制覆盖物方法的封装就完成后，只需要将 封装的这个方法TXMap 暴露出去，然后在 Vue 组件中进行引入，使用即可。

2、另外一个需求是在编辑的时候给项目在高德地图上选取地点，然后保存之后点击项目地点打开高德地图并定位到相应的地点。我们使用的是一个地图插件vue-amap。我们可以从后台获得我们想要功能的接口地址。

3、封装地图组件要求是：

（1）我们需要在创建组件实例的时候渲染HTML模板，因此需要将组件模板化；

（2）地图中的标记点marker是根据后端数据动态传入的，也就是说我们需要将数据源传递给Map组件，然后Map组件利用这个数据源将标记点渲染到HTML中；

（3）我们需要根据组件的业务需求，为组件设计api，例如为标记点添加信息窗体、自定义信息窗体内容。

我们是在当用户实例化组件时，需要传入容器的id名container和配置选项options（可选，不传则为默认配置项），注册地图实例并渲染到容器中。

因此我们设计了多个Api：

1 用户可以传入标记点数据markerList，渲染到地图中

2 为每个标记点marker注册信息窗体，并在点击标记点时弹出相应信息窗体

3 用户可以自定义窗体内容，传入信息窗体的标题title和内容content

4 为地图加入生命周期和点击事件，监听地图加载完成和点击地图事件

封装table表格

在父组件内定义表头名字（label）表名（key）type类型，请求接口url，method，以及传递的参数。通过父传子的方法传递到子组件，子组件开启watch深度监听deep,和默认初始化第一次监听immediate，调用合并父组件传递参数的方法，使用for in方法替换掉子组件data的数据（Object.keys(this.table_config).includes(key)），页面中使用v-if,v-else-if,以及v-else判断，如果是function类型则调用父组件内传递过来的回调函数，拿到参数。

如果是slot的话，使用插槽的方式，把参数传递出去。使用事件总线，子组件使用$emit把参数传递出去、父组件在created钩子里面使用$on来接收子组件传递过来的参数。

公司开发的管理后台系统需要用到大量的表格数据，每个页面内容大致相同，为了便于后期维护，便考虑将el-table进行封装，将公共部分内容提取出来，后期只需对表格内容进行维护即可。

把el-table中的属性数据显示都是用父组件传进来，都可以自定义。

属性：

data:表格数据

height:高度

stripe: 表格为斑马纹。

filter-multiple：数据过滤的选项是否多选。

方法：

row-class-name：行的className回调方法，用来给指定行进行颜色高亮区分。

sort-change： 当表格的排序条件发生变化的时候会触发该事件。

filter-change： 表格筛选条件发生变化的时候会触发该事件。

表格内容自适应屏幕宽高，溢出内容表格内部滚动。4.

表格数据操作 （单条数据删除、批量删除、重置密码、状态切换……）2.4

表格数据多选 （支持跨页勾选数据）4.2

表格序号、每行可自定义展开信息、表格头部自定义渲染 4.

表格列排序、单元格内容格式化（有字典会根据字典自动格式化）4.

树形表格展示（后期会增加懒加载）4.

表格数据导入组件、导出钩子函数 2.

表格查询（可携带初始参数）、重置功能的封装 1.

表格分页模块封装（Pagination）5.

表格数据刷新、列显隐、搜索显隐设置 3

比如说我在上一家公司 穷游后台管理系统 这样的一个项目中，我对表格进行的一个封装，技术栈是 vue2+element ui，在这个里面我主要是对el-table进行了二次封装，主要包括了表格的搜索区域，数据操作按钮区域，功能按钮区域，主体内容展示区域和分页区域

在搜索区域中，我主要是针对每个页面的搜索、重置方法都是一样的逻辑，只是不同的查询参数而已，所以我把表格的配置项columns提取出来，直接指定某个字段的 search:true 就能把该项变为搜索项，然后使用 SearchType 字段可以指定搜索框的类型，最后把表格的搜索方法都封装成一个钩子函数，这样做的好处是页面上就不会存在搜索逻辑了，实现页面展示和搜索逻辑的分离

我的数据操作按钮区域，因为表格数据操作按钮基本上每个页面都会不一样，所以我直接使用的是 作用域插槽 来完成每个页面的数据操作，作用域插槽可以将表格多选数据信息从我封装的table组件中以函数的形式传到页面上使用。

功能按钮区域，在这个区域里面，我主要封装了三个按钮，其功能分别为：表格数据刷新（一直会携带当前查询和分页条件）、表格列显隐设置、表格搜索区域显隐（方便展示更多的数据信息）。 可通过 toolButton 属性控制这块区域的显隐。

表格主体区域主要做数据显示，配置columns项传到我二次封装的table组件中，使用作用域插槽可以自定义每一列的显示自己需要的内容，还支持表格数据多选，然后内部都是使用函数钩子进行封装的

分页区主要设置的参数，当前页page，总条数totalCount，总页数totalPage，列表list

我把自己封装的这些东西整理成文档，供其他调用者查看

我认为组件的二次封装，就是把一个表格页面所有重复的功能，比如表格多选、查询、重置、刷新、分页器、数据操作二次确认、文件下载、文件上传等等都封装成可复用的函数，然后在自己封装的组件中使用这些函数。在页面中使用时，只需要把当前表格请求的 API数据，传给自己封装的组件，表格配置项 columns 就行了，数据传输都使用作用域插槽从自己封装的组件传给父组件就能在页面上获取到了。

form表单的二次封装

(1):拿下表单模板 (2):一个个的改造(文本，下拉，单选，复选等)
(3):添加按钮 (4):默认值的设定(修改表单的时候) (5):rules规则的处理
（1）在需要的组件中引入form表单组件，使用传值的方式，把需要的数据items传入到form组件中
（2）在同一个el-form-item中使用for循环遍历传过来的items
（3）使用v-if判断各个UI的类型（注意：Textarea的UI使用span来控制'行'）
（4）在写一个el-form-item引用按钮，使用for循环遍历buttons里面的数据
在这个按钮绑定单击事件传入三个参数一个所有的数据，按钮类型，一个回调函数，先判断按钮类型。如果是确定，调用submitForm函数并传入数据和回调
单击重置调用resetForm函数并传入数据
（5）先在父组件中使用default这个变量定义默认值，在created函数中写一个函数来初始化ruleForm里面的数据，在methods这个函数中写这个初始函数
注意：
（1）定义一个变量form,如果不定义这个变量，你在页面使用这个表单是填写不上任何东西

（2）如果是多选框默认值，在父组件可以使用数组来包裹默认值，在子组件中要定义一个数组，在进行判断，如果有这个default并且使用Array.isArray判断default.是数组使用concat合并新的数组，如果不是数组，就把这个default，push进新数组

（3）处理日期和时间格式，判断是否有这个默认值，如果有就使用new Date(default)传进来的时间格式必须加上日期，否则不显示

（6）rules规则使用集中式来校验（refObj.sync修饰符）

```
        // 校验表单
        validate(cb) {
            this.isValidate = null
            // 由于有多个表单需要进行校验所以直接遍历数据让来校验是否有没有填写的字段
            this.purchasingInformation.forEach((item, index) => {
                this.$refs['rulesForm'][index].validate((valid) => {
                    // vaild这个返回的数据如果当前index下标的表单有校验字段的内容为空就会返回false反之true，
                    // 这里通过这个来进行累加判断如果返回的有一个是false都是校验不通过
                    this.isValidate += valid
                    if (this.purchasingInformation.length === this.isValidate) {
                        // 如果都是所有的字段都填写了这时候通过校验callback返回给父组件处理
                        cb(valid)
                    }
                })
            })
        buttons: [
        {
            // 判断类型
            type: 'primary',
            label: '确定',
            action: 'submit',
            // 回调
            call: data => {
            // form为数据
            console.log(data)
            }
        },
        {
            // 判断类型
            type: 'primary',
            label: '重置',
            action: 'reset',
            // 回调
            call: () => {
            // form为数据
            console.log('reset')
            }
        },
        {
            // 判断类型
            type: 'primary',
            label: '取消',
            action: 'quxiao',
            // 回调
            call: () => {
            // form为数据
            console.log('quxiao')
            }
        }
    ],
```

导出表格

简单表格导出
为表格添加样式（更改背景色、更换字体、字号、颜色）
设置行高、列宽
解析 ant-design 的 Table 直接导出excel，根据 antd 页面中设置的列宽动态计算 excel 中的列宽
多级表头（行合并、列合并）
一个 sheet 中放多张表，并实现每张表的列宽不同

我有了解到两个方法，一个是xlsx，很好用，但是默认不支持改变样式，改变样式是需要花钱的，所以我又了解到ExcelJS，他就是集成很简单，可以简单的导出表格还可以添加表格的样式等，
我们想要把表格下载到本地还需要另一个库：file-saver

workbook：工作簿，可以理解为整个 excel 表格。
worksheet：工作表，即 Excel 表格中的 sheet 页。
我们可以通过 worksheet.columns这个属性来设置表头，通过row来添加一行或者同时添加多行数据，是我们在项目当中使用最频繁的属性

导出Excel

使用map方法定义好要导出得表头以及内容，使用 xslx.utils.json_to_sheet 把数据转换成 sheet 格式，xslx.utils.book_new 新建一个表格，使用 xslx.utils.book_append_sheet方法 把 sheet 数据插入xslx.utils.book_new 表格当中，使用 xslx.writeFile（外特fai屋） 导出表格。

导入Excel

上传事件使用async、await，使用FileReader.readAsBinaryString（）方法将文件读取成二进制字符串数据，用xlsx.read方法解析二进制数据，利用该对象得Sheets[(key值使用)sheetName[0]]拿到数据，通过xlsx.utils.sheet_to_json转换成json格式得数据。声明了一个对象定义了表头内容，以及type类型，通过forEach循环加for in循环判断，使用对象得hasOwnProperty（豪桑破rua破体）判断key值存在与否，判断值得类型，对应得key与value写成对象，添加到要上传得数组，然后把数据一条一条得传递到服务器。

websocket

WebSocket是一种在单个TCP连接上进行全双工通信的协议。

ws，wss，http，https都是属于应用层的协议，但是基于协议层 rtpc直播平台　　mqtt硬件　 xspp

ws：通讯协议
wss：对当前通信进行加密
http：基于TCP，rtpc，基于udp
https

基于TCP协议

五层网络协议：
    1.应用层 ： http，
    2.协议层 ： tcp/udp
        tcp:长连接（三次握手，保证传输的完整性），udp:短连接（无法保证数据的完整性 ）
    3.网络层 ： 通过系统封装，向外界发送请求
    4.数据链路层 ： 网卡
    5.物理层 ： 网线

区别

http具有局限性，单向，只能 客户端（client） 向 服务器（server） 发送请求
ws：双向，全双工通信 客户端 ->服务器 服务器->客户端

客户端：WebSocket
服务端：ws/socketio

流程

使用class类创建了一个单例模式，定义了一个链接服务器的方法，一个回调函数注册的方法，一个取消回调函数的方法，以及一个向服务器发送数据的send方法，在链接服务器方法内建立Socket的onopen链接成功事件，onclose链接失败监听的方法,在监听链接失败的方法内调用链接服务器的方法，还有onmessage接收服务器发送过来的数据的方法，在onmessage方法内，首先把服务器推送过来的数据，使用JSON.parse()方法把数据转换过来，判断回调函数是否存在，根据服务器返回的状态，使用回调函数call方法，把数据传递到组件。在组件内created钩子内进行注册初始化图片表的回调函数，mounted告诉服务器我使用的那个回调函数以及那种图标类型，在destroyed生命周期内取消回调函数。

Socket遇到的问题

页面刷新时候报错（报错原因socket还没有跟服务器链接成功，就发送了信息），写了一个开关，在往服务器发送链接的时候判断是否链接服务器成功。在onopen链接成功以及onclose监听链接的时候关闭了开关。

Umi

Umi用起来更简单一点，antd组件都已经配好了，它主要就是一个可扩展性，实现了完整的生命周期，并且很好上手。

dva

dva 首先是一个基于redux和redux-saga的数据流方案，然后为了简化开发体验，dva 还额外内置了 react-router和fetch，所以也可以理解为一个轻量级的应用框架。

开始公司是使用的dva，但是dva本身也是存在一些问题的，所以之后还是用的mobx

ts

什么是ts

TypeScript是JavaScript的加强版，它给JavaScript添加了可选的静态类型和基于类的面向对象编程，它拓展了JavaScript的语法。
而且TypeScript不存在跟浏览器不兼容的问题，因为在编译时，它产生的都是JavaScript代码。

ts和js的区别

TS 是一种面向对象编程语言，而 JS 是一种脚本语言（尽管 JS 是基于对象的）。
TS 支持可选参数， JS 则不支持该特性。
TS 支持静态类型，JS 不支持。
TS 支持接口，JS 不支持接口。

ts优势

TS 在开发时就能给出编译错误， 而 JS 错误则需要在运行时才能暴露。
作为强类型语言，你可以明确知道数据的类型。代码可读性极强，几乎每个人都能理解。
TS 非常流行，被很多业界大佬使用。像 Asana、Circle CI 和 Slack 这些公司都在用 TS。

泛型

简单来说就是类型变量，在ts中存在类型，如number、string、boolean等。泛型就是使用一个类型变量来表示一种类型，类型值通常是在使用的时候才会设置。

泛型的使用场景:
    可以在函数、类、interface接口中使用

interface和type的区别

（1）interface和type都是用来定义对象类型
    如果是定义非对象类型，通常推举type，比如Direction，Alignment，一些function如果是定义对象类型的区别
（2）interface 可以重复对某个接口定义属性和方法
    type定义是别名，别名是不能重复的，重复会报错
（3）type可以申明联合类型和元组类型

any和unknown的区别

会绕过类型检查，直接可用，而unkonwn则必须要在判断完它是什么类型之后才能继续用

unknown类型的值也不能赋值给any和unknown以外的类型变量

在联合类型中,unknown类型会吸收任何类型。即如果任一组成类型是unknown，联合类型也会相当于unknown；而any更是能吸收unknown类型，如果任一组成类型是any，则联合类型相当于any

B端和C端

c端

指的是消费者、个人用户Consumer;顾名思义就是面向个人用户提供服务的产品,是直接服务于用户的

b端

指的是企业或商家Business;顾名思义就是面向商家、企业级、业务部门提供的服务产品,是间接服务于用户的。

CRM系统和ERP系统

CRM系统

CRM系统是一个以客户为中心的专门用于管理与客户关系的软件系统，它确保与客户在销售、营销、服务上的每一步交互都顺利、高效，从而提升企业业绩。

CRM是企业最重要的数据中心，记录着企业在整个市场营销与销售的过程中和客户发生的各种交互行为，以及各类有关活动的状态，并提供各类数据的统计模型，为后期的分析和决策提供支持。

ERP系统

ERP系统是指企业资源计划系统。ERP是一个实现信息集成的管理系统，是企业内部的所有业务部门直接或者企业与外部合作伙伴之间交换和分享信息的系统。

ERP通过集成的系统和优化的流程，运用工作流让企业运行变得更加流畅，通过软件为载体，将整个业务流程转换到线上模式。从而使得企业运行资源利用更加高效。

首屏加载优化方案

1.路由懒加载
2.CDN（就近网络派发原则），挂载到CDN上加载
3.服务端渲染（Server-Side Rendering），是指由服务侧完成页面的 HTML 结构拼接的页面处理技术，发送到浏览器，然后为其绑定状态与事件，成为完全可交互页面的过程。
5.页面使用骨架屏

taro是怎么部署上线的

1.注册登录
　　在 微信小程序官网 注册账号，注册完成后进行登录
2.获取 AppID
　　AppID 作为一个唯一标识，在后面小程序开发里面起到贯穿始终的作用，注册登录进去之后，左侧 - 开发 模块，“开发设置” 里面可以获取到你的 AppID
3.项目开发
　　可以使用市场上已经存在的框架或者原生的小程序代码进行开发，下载 微信开发者工具 ，在微信开发者工具里面设置好 AppID ，项目编译之后运行在开发工具里面，没有问题的话点击 上传 即可把编译之后的代码上传到个人账号里面
4.项目上线
　　在 微信开发者工具 中上传的代码，首先进行基础的设置：应用名称 icon 等，然后点击提交审核，等待一段时间（半天左右），若审核通过，则可以把你审核通过的代码上传到线上版本

CI/CD 前端项目自动化部署

传统的 项目发布流程:代码提交到仓库中--->开发人员打包可以部署的--->发给运维OPs,部署到不同环境上--->test测试/压力测试可以自动测试，但是功能测试需要测试---->(有问题返回到开发)
流程：计划--->开发--->构建--->测试--->部署--->运营--->监控--->计划
CI:(主要操作)持续集成，在代码变更之后会自动检测、拉取、构建，（在大多数情况下）进行单元测试的过程
CD:持续交付，整个流程链，会自动监测源代码的变更并通过构建、测试、打包和相关操作运行他们以生成可部署的版本，给用户使用
CD:持续部署。直接把应用发布到生产环境中，

1. 项目环境变更：
   测试环境（数据量小，环境单一）--->预发布环境（数据和生产环境可能差不多）--->灰度测试--->生产环境
2. 持续继承的组成要素：
   版本控制器：github、gitlab
   构建脚本&工具
   CI服务器

echarts图表

1、柱状图通过option.series（C维斯）数组内第一个对象的type属性设置为bar。（注意：需要xAxis轴，与yAxis轴）

2、折线图通过option.series（C维斯）数组内第一个对象的type属性设置为line。（注意：需要xAxis轴，与yAxis轴）

3、散点图通过option.series（C维斯）数组内第一个对象的type属性设置为scatter（四台吹）。（注意：需要xAxis轴，与yAxis轴）

4、饼图通过option.series（C维斯）数组内第一个对象的type属性设置为pie。（注意：不需要X轴与Y轴）

虚拟列表

对可见区域进行渲染，对非可见区域的数据不渲染，以减少消耗，提高用户体验，是对长列表的一种优化

思路

（1）写一个代表可视区域的div固定器高度，通过overflow使其允许纵向y轴滚动
（2）计算可视区域中可以显示的数据条数（可视区域的高度除以单条数据的高度）
（3）监听滚动，滚动条变化的时候计算出被卷起的数据的高度
（4）计算可视区域内数据的起始索引，也就是区域内第一条数据。可以使用卷起的高度除以单条数据的高度可以拿到
（5）计算可视区域内数据的结束索引，通过起始索引加上之前计算出来的可以显示的数据的条数
（6）拿起始索引和结束索引中间的数据渲染到可视区域
（7）计算起始索引对应的数据在整个列表的偏移位置并设置到列表上

！！！！！ 其他！！！！！

react中状态提升

将状态属性提升到上级组件内管理

react受控组件

表单绑定状态属性

react数据不是双向的，没有vue中的v-model，需要使用value绑定状态和onChange中获取到当前输入的值，并把状态改变成这些值

参数

参数:修改的状态对象　回调函数
作用:修改状态 第二参数的作用:dom更新之后立刻执行的回调

高阶函数

一个函数接收一个函数作为参数，返回一个新的函数

常用的高阶函数

Promise、setTimeout、map、forEach、filter、every、some、find/findIndex、reduce、sort4

高阶组件

重用代码：提取出来逻辑，利用高阶组件的方式应用出去，就可以减少很多组件的重复代码

封装高阶组件思路

封装一个connect高阶组件，接收两个参数（这里指a,b函数），connct执行返回一个函数，（这里指函数c）,c函数为当前组件。使用上下文机制获取到store。调用getState获取公共数据，把a,b方法执行，并把公共数据作为a函数的参数，把dispatch方法作为属性添加给状态，通过store.subscribe监听当前组件的变化.最后把当前组件返回并传递给当前组件，组件就可以通过props获取到公共状态的所有数据和dispatch方法。

HOC

HOC就是高阶组件能够复用的逻辑，可以利用高阶组件实现一个换肤功能。
封装一个高阶组件，传递的参数为组件，在高阶组件中引入样式，因为在高阶组件中引入css样式的权重值会高于之前的。

作用：
- 代码重用，逻辑和引导抽象
- 渲染劫持
- 状态抽象和控制
- Props 控制

高性能组件

PureComponent

内部帮我们优化了shouldComponentUpdate的逻辑，当新的属性和状态都没有发生改变就不更新了，只有一个发生改变就返回true，不会多次引发render

PureComponent对比Component区别

相对于Component来说，PureComponent会默认设置一个shouldComponentUpdate周期函数，在生命周期内部对属性和状态进行'浅对比'
浅对比：只会对比第一层，拿nextState和this.state的第一层进行对比

PureComponent优缺点

优势：
　　不需要开发者自己实现shouldComponentUpdate就可以进行简单的判断来提升性能
劣势：
　　可能会因为深层的数据不一致，而产生错误的否定判断，从而shouldComponentUpdate结果返回fasle，界面得不到更新
　　解决：改变堆内存地址

登陆拦截流程

首先在定义路由的时候就需要多添加一个自定义字段 requireAuth，用于判断该路由的访问是否需要登录。如果用户已经登录，则顺利进入路由， 否则就进入登录页面。定义完路由后，利用 vue-router 提供的钩子函数 beforeEach() 对路由进行判断。每个钩子方法接收三个参数：to，from，next，确保要调用 next 方法，否则钩子就不会被 resolved。

组件化思想和模块化思想

组件化思想：把重复的代码提取出来合并成为一个个组件，组件最重要的就是重用（复用），位于框架最底层，其他功能都依赖于组件，可供不同功能使用，独立性强。
模块化思想：分属同一功能/业务的代码进行隔离（分装）成独立的模块，可以独立运行，以页面、功能或其他不同粒度划分程度不同的模块，位于业务框架层，模块间通过接口调用，目的是降低模块间的耦合，由之前的主应用与模块耦合，变为主应用与接口耦合，接口与模块耦合。
模块就像有多个USB插口的充电宝，可以和多部手机充电，接口可以随意插拔。复用性很强，可以独立管理。

线程和进程的区别

1.进程是资源分配最小单位，线程是程序执行的最小单位。 计算机在执行程序时，会为程序创建相应的进程，进行资源分配时，是以进程为单位进行相应的分配。每个进程都有相应的线程，在执行程序时，实际上是执行相应的一系列线程。
2.地址空间：进程有自己独立的地址空间，每启动一个进程，系统都会为其分配地址空间，建立数据表来维护代码段、堆栈段和数据段；线程没有独立的地址空间，同一进程的线程共享本进程的地址空间。
3.资源拥有：进程之间的资源是独立的；同一进程内的线程共享本进程的资源。
4.执行过程：每个独立的进程程有一个程序运行的入口、顺序执行序列和程序入口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
5.线程是处理机调度的基本单位，但是进程不是。由于程序执行的过程其实是执行具体的线程，那么处理机处理的也是程序相应的线程，所以处理机调度的基本单位是线程。
6.系统开销：进程执行开销大，线程执行开销小。
7.定义不一样，进程是执行中的一段程序，而一个进程中执行中的每个任务即为一个线程。一个线程只可以属于一个进程，但一个进程能包含多个线程。

精准匹配

exact

为什么在多级路由中不建议使用精准配

导致深层级路由无法匹配

withRouter

不受路由和管控的组件想要拿到路由信息，可以使用withRouter

作用

可以将一个非路由管控的组件包裹为路由组件，使这个非路由管控的组件也能访问到当前路由的match、location、history对象

原理

是一个高阶组件，接收当前的组件作为参数，将这个组件包裹进Route里面，并返回一个新组件，将路由信息对象match、location、history放进这个组件的props属性中

```
import { Route } from 'react-router-dom'
function WithRouters(Com) {
    return class WithRouters extends Component {
        render() {
            return <Route component={Com} />
        }
    }
}
export default WithRouters
```

路由跳转方式

声明式：NavLink，Link
编程式：this.props.history.push，this.porps.history.back

history.push()，hash.push()，history.replace()，history.go()

如何优化react程序

1:使用函数组件，减少性能消耗，
2:不使用ref或减少使用获取dom节点的操作
3:不操作dom节点
4 :使用purecomponent和shouldcomponentUpdate优化组件
5使用redux公共状态
6循环遍历key使用最好使用id不用下标

react核心库

```
react :核心库，JSX元素，状态，属性..., 例如如何创建虚拟dom
react-dom : Dom渲染库，和真实dom相关的代码 ( render函数：把虚拟dom（JSX）转义成真实的dom)
```

redux取消subscribe监听

```
subsrcibe本身有返回值，返回值为取消监听函数
```

react路由建议倒叙的原因

```
便于Switch筛选
```

伪数组转数组的方法

```
Array.from(),[...伪数组],for循环
```

Commonjs导入的方式

```
导入：require(),导出：module.exports={}
```

node核心模块

```
fs，http，url，path，querystring
```

express常用函数

```
express.redirect()，express.static()，express.json()，express.urlencoded()
```

浏览器中可以发起http请求的方式

```
地址栏url路径，img标签src属性，link标签href属性，a标签href属性，script标签src属性，from表单提交,ajax
```

from表单的验证方式

```
1.表单上加rules｛object｝这种方式需要在data()中写入rule{},对于需要校验字段prop中的如visitorName写上验证规则。
2.在el-form-item单个添加。
3.动态增减表单项
```

webpack中如何配置多入口打包

```
path:输出文件的路径
filename：输出文件文件名

entry: {
    app: 'index.js',
    build: 'build.js'
},
output：{
    path： path.join(__dirname, './dis''),
    filename: '[name].js'
}
```

$route 和 $router 的区别

```
1.$router是VueRouter的实例对象，这个对象中是一个全局的对象，他包含了所有的路由，包含了许多关键的对象和属性
2.$route是一个跳转的路由对象，每一个路由都会有一个$route对象，是一个局部的对象，可以获取对应的name，path，params，query等
```

vue-router两个内置组件

```
<router-link></router-link><router-view></router-view>
```

Vue路由跳转方式

```
声明式跳转<router-link></router-link>
编程式跳转this.$router.push()或者$router.replace()
```

Vue路由封装流程

```
下载vue-router插件，创建router文件，引入vue和vue-router模块，调用vue.use(vueRoute)
创建vueRoute实例 实例里的routes属性写入路由信息，导出这个实例
在main.js文件中引入router文件导出的实例并注册，路由配置：{routes}
```